On Offloading Control Plane Applications to the Data Plane

A Master's Thesis
Submitted to the Faculty of the
Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya

presented by

Albert Gran Alcoz

In partial fulfilment
of the requirements for the degree of

Master in Telecommunications Engineering

accepted on the recommendation of
Prof. Dr. Laurent Vanbever
Prof. Dr. José Antonio Lázaro

Eidgenössische Technische Hochschule Zürich

Networked Systems Group, 2018

To my parents

Abstract

Scheduling is one of the main active players in the quest for programmable networks. Despite the numerous research efforts that have been dedicated in latest years, not a single scheduling framework has resulted to be powerful enough to outperform the rest in a wide variety of scenarios. A new perspective to the problem has been therefore recently brought up, which suggests abandoning the pursue of a global scheduling solution, and moving into a more flexible and programmable conception. Network equipment should be designed to support different algorithms, from which it could select and configure the most appropriate one at each moment to face the instantaneous requirements of the dynamic nature in traffic demands.

With the idea of making scheduling more programmable, new abstractions have been already defined, based on decoupling the process in two steps: a programmable-pipeline determining the order in which packets should be transmitted, and a fixed-logic push-in first-out (PIFO) queue draining packets in the desired arrangement. While PIFO abstraction is innovative and deeply promising, its hardware implementation is not straightforward. To the intrinsic difficulties of such a complex queuing design, adds the fact that ASIC production is by definition a multi-year process, propelling the release of a hardware built-in PIFO too far from expectations.

Aiming to fill this temporal problem, in this thesis, a novel approach is proposed. Would it be possible to achieve a PIFO-behavior with the current resources available in nowadays networks? By trying to answer this question, we will embark in a journey that will span from revising the first quality of service proposals introduced at early networks, to discussions on how to reach predictability in potential future generations. Altogether, with the focus centered on squeezing the maximum benefit from the recent advances in network programmability, with special emphasis in the latest proceedings for programmable forwarding data planes.

Acknowledgments

First and foremost, I would like to express my deepest thanks to Professor Laurent Vanbever, for having given me the opportunity to be part of the Networked Systems Group during these months. For having believed in me since the very first day, and for all the time devoted, inspiring advice and endless support throughout all the project. You have been a reference for me, and working with you during these months has been a pleasure and an immense privilege. I would also like to thank Professor José Antonio Lázaro, for having been always there and for all the efforts in trying to make this stay possible.

To the NSG fellows. Thank you Edgar, for all the insights, discussions and late-night talks. To Thomas, for the wise words and reflections towards my professional career, for all the *holycows* and *uni-mensas*; to Rudi, for the permanent good humor and positive vibes; and to Roland, Maria, Tobias and Ahmed, for having opened all the doors to me. I have always felt as one more in the family, and I will be forever grateful.

To the best office-mates, Max, Andrea, Amine, Ferdinand, Alex and Markus. All those hours locked in the lab would not have been the same without your company. Thanks for sharing this experience with me. Let's keep pushing.

I would also like to thank everyone who has brought this experience far beyond the academic world. Sergi Caelles, for being my big brother during these months. Vas ser tu el que em va fer conèixer l'ETH, el que m'ha guiat professionalment. M'has ajudat a re-encendre la flama de voler seguir aprenent cada dia, apuntar el més amunt possible i escollir sempre el millor camí. Estic orgullós de tenir-te com a amic. Pablo, Yash, Yashveen, John, Guus, Daniel, Jay, thanks for being the best roommates I could have never asked for.

Thanks to all my friends and family back home, for the continuous care and best wishes. And finally, to the two most important people in my life, my parents, tot el que sóc avui és gràcies a vosaltres, no em cansaré mai de dir-vos que, no hi ha paraules que expressin tot el que us estimo. Aquesta tesis és per vosaltres.

Contents

1 Introduction

With the Internet growth in the late 1990s, a wide range of services and applications began to appear, and started sharing the limited resources of the networks. Each of these applications had been designed for a completely different purpose, and therefore came accompanied with a particular set of requirements in terms of mainly bandwidth, latency and jitter. This broad diversity of applications with distinctive demands, encountered a highly decentralized network that was being governed by a non-guaranteeing 'best effort' policy. A new challenge for the research community had been born: how to optimally distribute the finite network resources throughout the different application flows, so that they could maximize their performance demands. And, in case not all the traffic objectives could be satisfied, which services should be chosen to be prioritized.

During the following years, and up to today, the field has been widely studied, with several proposals presented, and even some of them implemented. The first attempt came under the umbrella of Internet Engineering Task Force (IETF) Integrated Services architecture (IntServ), in which application flows were prescribed to perform individual reservations of the resources intended to use, before they could gain finally access. Several problems of scalability, derived from the fact that each router in the network had to be compliant with the protocol, together with the amount of state required to be stored, and the difficulty in keeping track of all network wide reservations, made this architecture not practical at the end. It was then the turn of differentiated services (DiffServ), a highly simplified QoS architecture, aiming to provide flows personalized treatment by just classifying them under a 6-bit Type of Service range. DiffServ is what most of nowadays network equipment support, but, as the number of applications has kept increasing, it has become more evident that a finer granularity protocol is still highly needed. While those two proposals have been the ones reaching further in terms of deployment and implementation, a great number of alternatives have been also suggested and deeply discussed by the academia. After all the research efforts, however, nowadays most services still use the de-facto best-effort model, evidencing the poor success that the various presented options have achieved. QoS remains therefore an open problem that, regardless the great advance that network engineering has experienced in the latest years, and the amount of effort devoted by the research community, is still waiting to be properly solved.

However, recent advances in network programmability, together with the increased flexibility of forwarding data planes, seem to suggest that all the tools required to face the objections of such an urgent challenge, have been finally made available. It might be now the appropriate moment to make a definitive attempt in trying to find a robust solution to the problem.

When thinking about the next generation of networks, one would like to dream on intelligent systems capable of distributing their resources among the instantaneous demands of various flows. Doing it, with complete awareness of the results that a particular allocation would suppose. Understanding what each decision would mean in terms of quality of experience for the

end-users, and being able to quantify the costs and implications of a given arrangement. One, that would dynamically adapt and rapidly react to changes in configurations and performance objectives. An intelligent system that could know how much resources would it require to achieve, for instance, a certain queuing delay, or which levels of utilization could optimize the threshold between efficiency and congestion. Such a system that, in result, would be able to predict, plan and accommodate guaranteed services to demands, would be the one finally driving us to a new world of what we could call predictable networks.

This thesis is about that. With the focus put on bringing the next grain of salt in the path towards predictable networks, we will work on improving QoS management in current networks, starting from one of its main pillars which is scheduling. Following the document structure, in Chapter 2 we will overview the state-of-the-art in scheduling algorithms, and revise the evolution of the different techniques, analyzing their best performance scenarios and principal drawbacks. By going through the various proposals, we will observe that there is not a master scheduling algorithm, outperforming all the others in all situations. That will bring us to believe that networks will have to support and adapt schedulers according to their needs, which will drive us to the idea of making scheduling programmable. At the same time, we will see that scheduling has not taken part of the data plane programmability revolution, and we will discuss what techniques should be required to make it possible. In Chapter 3, our new approach for facing scheduling programmability limitations will be presented. By using strict priority hardware, intrinsic in most of current network switches, we will implement a push-in-first-out queuing approximation that will allow us to deploy a broad range of programmable scheduling algorithms. Chapter 4 will focus on evaluating the proposed solution, with detailed simulations to illustrate the main characteristics but also demonstrating its limitations and disadvantages. A variety of performance objectives, from flow completion times to minimizing tail delays, will be tested and accurately verified. Finally, Chapter 5 will be left for discussions on our proposal, further steps to be considered, as well as future challenges and opportunities.

2 What About Predictable Networks?

2.1 Introduction

Recent years have supposed a revolution in networking. The first change of paradigm came with Software Defined Networking (SDN) and the idea of decoupling the network control plane from the data plane, giving a broader flexibility in the management and network operations. The global view of the topology could be used from a controller to optimize the network functions, distributing those policies on the forwarding plane through protocols like OpenFlow [MAB+08]. With SDN, however, all the programmability was being centralized at the control plane, implemented in software, leaving the data plane as a group of hardware-based forwarding equipment limited to obey the guidelines received. Some years after, diverse people started to consider the benefits that giving also programmability to the data plane could provide. The distributed nature in which the Internet had been created, together with the vulnerabilities that supposed having a unique centralized network brain, seemed to suggest that giving more flexibility to the data plane could offer a new spectrum of opportunities. That is how, after some years of progress, data plane programming languages started to appear. In 2014, Programming Protocol-independent Packet Processors (P4 programming language) was finally proposed [BDG+14], together with the Protocol Independent Switch Architecture (PISA), a framework providing the basic abstractions for the definition of programmable data plane solutions. With them, a new generation of programmable network switches, into which specialized programs could be built, defining manners in which packets should be treated, and run on top, started to be developed by the different silicon industries. Those new hardware equipments allowed rapid iterations between protocol design and implementation, as the usually-multi-year-process of creating a new fixed-function equipment could be reduced to just writing a new program and pushing it to existing architectures. And, contrary to the conventional belief, programmable switches showed that they could not only reach, but also improve the performance of fixed-function switches. With a rate of 6.5Tb/s Barefoot Tofino was proved to be the fastest switch in the world, with the same cost and power than traditional fixed-function chips, and with the greatest advance of being completely programmable through software written in P4 [McK17]. Some of the new functionalities derived from this new paradigm, like having access to application-and-above layers to perform forwarding, or the ability of saving state information in each switch along the path, made clear that, with data plane programmability through platforms like P4, an immense range of new possibilities had definitely been released.

That is also how this thesis started. While aiming to look for which applications, traditionally implemented in a centralized manner, could be interesting to offload into a distributed philosophy through programmable data planes, it resulted quite evident that the most urgent one was quality of service. Recent studies had envisioned, that under a centralized environment with knowledge of the traffic, forwarding of packets could be optimally arbitrated to avoid con-

gestion and maximize traffic performance [POB+15]. Although such a centralized architecture could be easily perceived as impractical, the promising results motivated our idea of trying to move towards predictable networks. And with the latest results in data plane programmability and the possibility of offloading solutions into distributed scenarios, we started to explore the elements that our concept of predictable networks should explicitly include.

Predictable networks can be defined as those systems able to anticipate, plan, adapt, update and react to the continuously-changing and highly-diverse traffic demands. To sustain that the idea of predictable networks is actually feasible, there is the fact that network behaviors are, by nature, continuous over short periods of time. With high probability, the traffic that a network experiences during a small interval of time (less than a second), will be highly correlated to the one experienced in previous and following seconds. Application flows usually act in a repetitive manner over a short span of time.

Achieving predicting behaviors, however, is not expected to be an easy-task. Throughout this first chapter, we will propose, discuss and analyze the major players that, in our view, should be primarily considered when trying to work towards this new paradigm. The first ingredient that should be deeply understood is the nature of traffic running on top of nowadays networks. We can find different type of connections, with a remarkable range of performance objectives. In order to be capable of defining techniques aiming to face those objectives, it is clearly needed to understand them correctly in advance.

- **Traffic types and requirements**: Application flow demands can usually be reduced into specific amounts of throughput, delay and jitter, although each metric can be expressed in finer demeanors. Throughput can be defined as the rate of bits successfully transmitted during a specific interval of time (this is, the bits transmitted during the time period [t, t + T), divided by the time T). Usually the word bandwidth generates confusion with the throughput. The bandwidth, also called capacity, is the maximum rate at which a sender can transmit data along a link, and it is an intrinsic physical property of the wire. However, it is closely tight to the throughput, as a higher bandwidth share allocated to a flow, means that this flow will be able to achieve higher throughput in its transmissions. The relation between the link bandwidth, and the aggregation of flow throughputs sharing the link in a specific moment, is the called link utilization.

  The delay or latency of a bit (or a packet) is the time required for the bit (or the packet) to go from source to destination. In other words, is the difference in time between the instant in which the bit reaches the destination, and the instant in which the bit departs from the source. This delay or latency can be divided in different contributions: the propagation delay, that is the time it takes for the signal to travel throughout the link, the processing delay, which is the time that it takes for the routers/switches to process the packet headers, manage memory, and select the appropriate output ports, and finally the queuing delay, that is the time spent by the packet in the queues, waiting for previous packets to be served before being finally transmitted. Other metrics of higher order, derived from the delay, can be the packet transmission time (i.e. the time that it takes the sender to transmit all the bits forming a packet), or the flow completion time, that is the amount of time that it takes to complete a transmission of a group of packets forming a flow or connection. As can be seen, requirements specified by using these metrics can be easily converted to their equivalents based on delay.

  Finally, the jitter describes the variability in delay, as perceived by consecutive packets on a flow. Supporting big amounts of jitter, for example in playback applications, requires

big-sized buffers in reception to store all the packets arrived, before they can be correctly reproduced at the adequate rate.

When treated individually, it is not extremely hard to know the tools and components available today that should be used to answer those requirements with a solid amount of guarantees. For example, in order to achieve low delays, it is required to separate big flows (also called elephants) from small ones (or mice), so that the last ones are not blocked in queues along the path and they can finish transmissions the sooner possible. Moreover, queues have to be designed with very little buffer size, so that they can stay almost-empty and buffer-bloats are not produced. For this reason, queuing disciplines and buffer management techniques are extremely important for delay. Concerning bandwidth, on the contrary, it is important to perform a proper selection of paths, with the use of routing protocols designed to provide quality of service or creating virtual circuits from the source. Mechanisms like load-balancing or multi-path selection can be used to efficiently distribute flows throughout available link resources. In reference to the queues, the solution for throughput is opposite to the one for delay. Long-buffered and deeply-filled queues are the best option to guarantee high levels of utilization for output links. It looks therefore, that throughput and delay have almost orthogonal requirements. An optimal solution should try to isolate them as much as possible, emphasizing the protection of short flows with very tight delay constraints.

For jitter minimization, traffic shaping or conforming is required. As it will be seen in Section 4.1, most scheduling algorithms are based on prioritizing some packets over the others. Those algorithms usually suppose a big source of jitter, as packets need to wait for others to be served, and the number and type of packets encountered in switches is not constant throughout the path. Not performing any type of scheduling (i.e. using the default FIFO behavior of queues) is the best solution in terms of jitter. But again, FIFO queues play against the requirements of delay, making jitter and delay almost orthogonal requirements as well. Regarding path selection for jitter, the best results are achieved when stability on the network is maintained. Intuitively, this means that all packets of a flow should try to be forwarded through the same path, to maximize the chances of finding the same congestion conditions, minimizing this way their difference in delays. Path stability also favors the fight against reordering needs, a very important factor to be considered in the design. Summarizing, dynamically adapting the routes to achieve better paths in terms of delay and throughput, if done at a very high rate, can play completely against in terms of jitter.

As can be clearly seen, delay, throughput and jitter requirements are quite orthogonal from each other, and although individually can be treated with the correct design of networks and tools, the challenge comes when various of those demands have to be treated together. In some restricted environments, it may be possible to draw orthogonal paths trying to isolate traffic of different demands. But that is not usually the case in common scenarios. When flows with different requirements have to share resources in the network, the question of how to optimally distribute them to maximize performance arises. In very few cases, it will be possible to guarantee service to all entering flows. In a wider range of situations, the network will have to choose (i.e. prioritize) which flows from all the candidates will be the ones achieving guaranteed quality of service, and which ones will have to be driven under a best-effort approach.

A good design for a predictable network should consider two types of requirements: hard-requirements and soft-requirements. Hard-requirements are those which need to be strictly satisfied in order to guarantee a correct performance of the service. Those are

the requirements that the network has to prioritize above all. A clear example of such requirements can be found in voice services. For a conversation to take place, it is needed to guarantee a minimum of 30Kbps throughput (assume fore a given codec, overhead definition, and sampling rate), and a maximum of 30ms of jitter, 150ms of one-way delay and 1% of packet drops. With a lower service than that, the conversation would be not acceptable and the service would become useless. In this case, the resources allocated would be completely wasted, so satisfying the hard requirements is of crucial importance to be sure that the service will, at least, take place in a correct manner. Hard-constraints are given as detailed magnitudes. For example, a critical delay could be defined as, at most, 1ms for an urgent application, or 5ms for a more-relaxed one.

Soft-requirements can be seen as just resource extensions to make the service better. Once having covered the hard basic needs, soft-requirements enhance the user experience. Those are not the biggest priority, but once all the hard requirements have been met, the network should consider how to distribute the remaining resources to satisfy application flows in a fair manner. An example of a soft-requirement for a file transfer protocol would be to maximize the throughput. It is not crucial for transferring the file to have a very high throughput, but doing it will make the transmission finish faster, which will be received with satisfaction from the end-user side. Once the basic demands have been covered, soft-requirements enter to action. They are given as maximizations: maximizing throughput, or minimizing flow completion time. With this distinction, the network task can be described as an optimization.

For all flows in the network, with an origin-destination tuple clearly specified, and a set of hard and soft constraints, assuming a clearly defined and stable network topology, the objective is to minimize the difference between targeted performance and the average result finally achieved.

Formally, let there be a set of flows $f \in F$, in which each flow is constrained by a hard performance target, p_h, which can be decoupled in delay, throughput and jitter requirements $\{d_h, t_h, j_h\}$, plus some softly-desired demands $p_s = \{d_s, t_s, j_s\}$. The network objective is to minimize the difference between target performance and the average performance resulting from the allocation resources received, $p_a = \{d_a, t_a, j_a\}$, while guaranteeing that the hard-requirements are strictly satisfied.

$$\begin{aligned} \min_{p_a} \quad & \Delta p_{s-a} \\ \text{where:} \quad & \Delta p_{s-a} = p_s - p_a \\ \text{s.t.} \quad & p_a > p_h. \end{aligned} \tag{2.1}$$

The optimization can be decoupled in three different optimizations, each concerning to one of the previously-defined requirement types.

$$\begin{aligned} \min_{d_a, t_a, j_a} \quad & \Delta d_{s-a}, \Delta t_{s-a}, \Delta j_{s-a} \\ \text{where:} \quad & \Delta d_{s-a} = d_s - d_a \\ & \Delta d_{s-a} = t_s - t_a \\ & \Delta d_{s-a} = j_s - j_a \\ \text{s.t.} \quad & d_a > d_h \\ & t_a > t_h \\ & j_a > j_h. \end{aligned} \tag{2.2}$$

As a conclusion, a predictable network can be defined as a one-block system, with a clearly described input and output.

  - Input:
    * Workload (set of flows with requirements)
    * Topology of the network (assuming stability throughout the optimization: no link failures, no capacity variability)
      · Links with capacity
      · Switches, specifying if programmable and the set of queuing disciplines supported
  - Output:
    * Circuit building through tagging mechanism
    * Instantiation of queuing disciplines
    * Such that workload achieves QoS requirements

- **Centralized vs. distributed approaches**: Interesting debate has recently arisen, discussing that the quality of service challenge has probably appeared due to the distributed nature of networks, in which decisions need to be taken at the local perspective of individual routers or end-hosts, loosing the sense of control over the packets experience throughout their journeys. Two type of philosophies appear: the ones thinking on a centralized solution to overcome those difficulties, and the supporters of a distributed approach, strongly moved by the recent advances in data plane programmability, specially in terms of saving-state capabilities and dynamic packet processing, who believe that recovering control in the inherited distributed nature is still possible, allowing us to keep the scalability, flexibility, and fault-tolerance advantages that have defined the way networks have been thought and brought where they are.

- **End-to-end solutions vs. critical switch**: Another consideration is the one between the strict necessity of an end-to-end framework or a plug-an-play solution that can be effective even implemented on a single switch. It is clear that, in terms of performance, end-to-end solutions will theoretically provide more robustness and therefore better performance results. However, if the change requires replacing existing infrastructure, complex configurations or exigent user contribution, then the solution is with high probability not going to succeed. A fair reasoning is to dream about an end-to-end solution as a long term objective, but realistically focusing on a solution design to be implemented on progressive steps, and where the benefits can be realized even if it is only adopted on some of the network most critical switches. Any valid idea should be ambitious enough to work at an end-to-end level, not only taking action on every switch, but considering them all in a global manner. Again, when gathering control of the whole network, an immense range of possibilities emerge. By taking and structuring the network as one, it is possible to not only think of working with queues and schedulers in a single switch, but on carrying information between consecutive switches to perform network-wide optimizations throughout the path. This can be combined with monitoring and network telemetry to dynamically configure the best schedulers at each hop, or adopting congestion control mechanisms. Even more, routing could also contribute by determining the optimum links to use according to their capacities and utilization. Load-balancing could also enter and, with it, multi-path techniques. Dreaming big is easy, and sometimes is needed. Although the idealistic solution should be always kept in mind, any objective can only be fulfilled when sufficiently small steps are taken in a progressive manner.

- **Key players and techniques**: Achieving predictable networks requires precise puzzling of different components. Everyone could increase the complexity as much as desired when thinking about the perfect solution. But, as considered in the previous lines, sometimes is important to focus on the principal actors taking the biggest part of the stake. Scheduling and path allocation are, without any doubt, the two big players in quality of service. In this thesis we will mainly focus on scheduling. The reason behind is the sense that path selection can depend on the scheduling algorithms supported around, and the background idea of merging route selection with dynamic configuration of programmable scheduling throughout the path.

Being scheduling the crucial point towards predictable networks, our first objective is the analysis of different algorithms that have been implemented until now, to learn which are the challenges they try to solve and what are their advantages and weaknesses. These learnings will allow us to describe the best scenarios in which they can be used, which will be fundamental in the future to achieve predictable networks. At the end, not only will it be important to understand which queuing mechanisms should be used to achieve each type of quality of service, but also determining correct ways to combine them, not having to depend on a global algorithm for all situations, but designing combinations of them for maximizing performances in different scenarios.

2.2 Evolution of scheduling algorithms

Scheduling just consists on determining the order in which packets should be transmitted at each switch or router throughout the network. And even being a super small piece of the whole QoS provisioning field, it has been extendedly researched since early days[1]. The first and most common algorithm is **First Come First Served (FCFS)**, in which the order of arrivals defines the resource allocation. All packets share the same buffer, with the packets arriving first being also the ones departing first. If the buffer gets full, new coming packets will be discarded. The main problem of FCFS is the non-guarantee of fairness between defined entities (for instance flows or users). A flow transmitting at high speed can capture an arbitrarily high fraction of the available bandwidth, or even block the transmission of other flows. A set of different FCFS queues, can work together under a Head of Line (HoL) Priority Queuing regime. A priority scheme is defined as the one in which low priority queues can only transmit when higher priority queues have finished draining all their packets. The same philosophy can be extended to separate the treatment of different types of traffic (e.g. real time and non-real time).

In 1987, J. Nagle proposed **Fair Queuing (FQ)** [Nag87], an algorithm that consisted on maintaining a separate queue for the packets coming from different instances, and serving those queues in a round-robin manner. This mechanism would prevent malicious sources to send packets at a too high throughput, which would just increase their personal queue. Although the idea of fairness was of crucial importance, the original FQ had several flaws. It did not consider that in a scenario with different packet sizes, although serving packets from different sources in a round-robin manner, big-size packets would still achieve higher throughput than smaller packets from other sources in the same round.

[1] Scheduling is just one more technique, that added to buffer management, congestion control and traffic engineering, form the basis of quality of service. If scheduling by itself has been a widely studied area, with numerous contributions since early times, one can clearly note that a detailed study of the QoS approaches and techniques would demand far more time and effort than the sufficed in this document. Curious readers are encouraged to follow materials in [Zho18] for a more extended study on those matters.

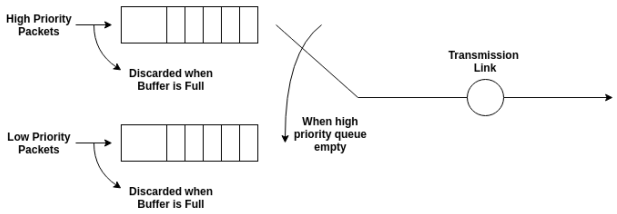


Figure 2.1: Strict priority scheduling.

Some other possibilities started to be explored. In **Weighted Round Robin (WRR)** [Sem01], the technique was similar to traditional Round Robin [HG86], with the main difference that each queue was given a certain weight. The way to proceed was just to normalize those weights so that they could become integers. The scheduler would then iterate in a Round Robin fashion among queues, serving as many packets as the converted-to-integer weight specified. This technique maintained max-min fairness only if the packets had the same size. In case the packets had variable size, then the weight-per-byte needed to be computed by dividing each queue weight by the average packet size of the queue and then this weight-per-byte could be finally converted to integer and proceeded in the same manner (serving as many packets per queue as their normalized weights specified). However, this required the knowledge of the mean packet size per each session, as a dedicated queue per session was being assumed, making complex its implementation. To solve the challenge of needing the mean packet size in advance, **Deficit Round Robin (DRR)** [SV95] was later proposed. The idea behind DRR was to set a counter per each queue that would track the amount of bytes sent by each flow. At each round, all counters would be set to a configured amount, defining the number of bytes that could be transmitted. Packets would be sent until this deficit was spent. Remaining packets would have to wait until the next round, when the deficit could be increased again. Those packetized versions of RR and DRR were able to achieve fairness in the long term. With the idea of providing instantaneous fairness, **Fluid Flow Fair Queuing** or **Generalized Processor Sharing (GPS)** [PG93], assumed that traffic arrived in a fluid manner and the scheduler rotated the queues while serving an infinitesimal amount of information from each queue at each round. It was obviously just a theoretical hypothesis, as such a fluid model was not implementable, but it has served until today as basis for a wide range of packetized algorithms, which try to approximate this fluid fair behavior.

In 1989 Shenker et al. presented **Weighted Fair Queuing (WFQ)** [DKS89], also called Packetized-GPS (P-GPS), an improved version of FQ which simulated a hypothetical bit-by-bit round-robin fashion. The round at which the last packet of each flow would be transmitted if the bit-by-bit round-robin technique was followed, was computed per each packet in the flow. Packets were then ordered in increasing finishing-round value (or virtual finish time). Whenever a packet finished its transmission, the following one would be the one with the smallest finishing-round value. With this mechanism, they attempted to provide isolated and equitable access transmission bandwidth, preventing high rate flows from taking all the resources. Two different possibilities were allowed according to preemption. The preemptive version, in which if a new packet with smaller finishing-round value arrives while a bigger finishing-round value is being transmitted, the transmission should be stopped to give the opportunity to the new-coming one. They also presented the possibility of giving more promptness (less delay) to flows utilizing less than their fair share of bandwidth, by including a new variable when computing finishing-round values.

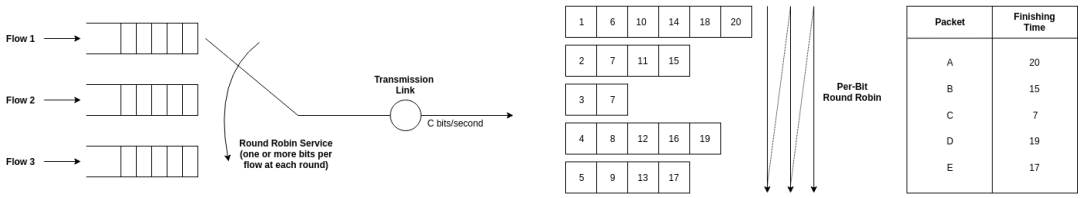


Figure 2.2: Finishing time computation in Weighted Fair Queuing.

Weighted Fair Queuing was specially designed to allow slow-speed links to provide fair treatment to different flows of traffic. By prioritizing low-bandwidth traffic over high bandwidth one, malicious input streams could not starve other flows of link bandwidth by generating large amounts of traffic, as that would not provide the requested portion, but just harm his own transmission until the requested bandwidth was decreased. This way, WFQ forced each flow to transmit at a fair share of the available capacity. Weighted Fair Queuing was fair in the sense that its departure times, and the ones in Fluid Flow Fair Queuing remained bounded by the maximum size of the packet divided by the output link capacity. The biggest complain to this algorithm was, however, that each flow required its own queue, which would suppose numerous memory references, making them unfeasible for operating in high speed networks. Even today, this type of approximations is rarely seen implemented in core networks, as the number of flows is very large (potentially thousands) and maintaining the state information for all of them supposes a significant level of complexity.

In 1990, **Stochastic Fair Queuing (SFQ)** [McK90] was presented, as a probabilistic variant of Shenker's algorithm to diminish its computational complexity inherent from the switch need of mapping source-destination pairs to a certain queue on a per-packet basis. In SFQ, the use of a hash function was proposed to easily map source-destination address tuples into fixed sets of queues. By periodically perturbing the hash function, the likelihood of hash collisions occurring during consecutive time intervals was substantially decreased. The general challenge remained the need of computing virtual finishing times, which required keeping track of the state for all backlogged users, posing a problem in terms of the real time computations. Further mechanisms to achieve fairness with smaller levels of complexity were therefore still required.

New algorithms like **Self-Clocked Fair Queuing (SCFQ), Start time fair queuing (STFQ)** and **Worst-case fair weighted fair queuing (WF2Q)** [Sah08] [Gol94] presented alternative ways to make less tedious the computation of termination virtual times in WFQ. However, the price to pay was a reduction in the obtained fairness. A discussion on the main differences in their implementation details can be found in Section 4.3. Other techniques, such as resource reservation algorithms, also called reservation services, in which connections were forced to negotiate with the network resource scheduler the maximum and average throughputs at which they were allowed to transmit, and providing best-effort services to the rest of connections by using the remaining bandwidth counterparts, were also deeply discussed.

With the rise of modern commercial operations, such as web services, cloud computing and storage, new challenges needed to be faced. In particular, data-centers experienced that most of their applications (financial services, social networking, recommendation systems and web search) often had very demanding latency requirements. Commonly adopted fair sharing schemes were far from optimal in terms of latency. This was mainly because latency-sensitive

flows often got queued up behind bursts of packets from large flows of coexisting workloads (like backup, replication, data mining etc.). Several protocols and algorithms were therefore presented trying to decrease the flow completion times in those scenarios, the called real-time schedulers. It is important to remark that in data centers the majority of the flows are mice (reduced sized with strict latency requirements), while only a few are elephant (throughput requirements). Traditional real-time scheduling techniques like **Earliest Deadline First (EDF)**, which focused on minimizing the number of missing-deadline (late) flows and **Shortest Job First (SJF)**, which minimized mean flow completion times, adopted priority queuing schemes to give preference to flows in terms of deadline (tighter deadline first) and flow sizes (short flows first) respectively. Their application in data centers was not completely straightforward because of the following reasons:

- They were presented as centralized algorithms, which meant that they required global knowledge of flow information.
- Prioritizing flows ideally would require each of the concurrent flows to have a unique priority class. Data centers could allocate thousands of flows, while modern switches only supported around 10 priority classes.
- They assumed preemption, allowing newly arriving tasks with smaller deadlines to be completed before already scheduled tasks of lower priority levels.

Still in the group of real-time schedulers, **Shortest Remaining Processing Time (SRPT)** was demonstrated to be the optimal algorithm for minimizing average FCT when scheduling over a single link. By prioritizing flows with least work remaining, it had the main advantage of achieving per-packet granularity, something that SJF had not been able to do. Its principal drawback, however, was the lack of consensus on how to optimally specify the deadline of each packet. Several recent algorithms based on SRPT, like pFabric, suggest using the received TCP acknowledgements to estimate these deadlines, and trying to make use of new programmable switches to update them as packets travel along the path.

In 2011, $\mathbf{D}^3$ [WBKR11] proposed a new technique to meet flow deadlines, in this case by following a first-come first-reserve (FCFR) basis. Every arriving flow to the switch would request a certain rate, where $rate\_requested$ is defined as $flow\_size/flow\_deadline$. If the rate requested was available in the output link, the switch would allocate it to the flow. D^3 results depended highly on the flow arrival order, and its performance was still far from the desired.

With the idea of moving the complexity of priority scheduling to the end host, **Preemptive Distributed Quick flow scheduling (PDQ)** **[HCG12]** was presented in 2012. To avoid the problem of having a large number of flows and not enough queues to distinguish them all, it proposed adapting directly the flow transmission rates. By controlling each flow's sending rate, PDQ regulated the traffic to retain packets from low-priority flows at senders, being able to achieve finer priority granularity with only FIFO tail-drop queues in the schedulers along the path. PDQ was presented as a distributed protocol instead of a scheduling algorithm, and was able to approximate a range of scheduling disciplines, specially designed to adapt traditional real time schedulers (EDF and SJF) to the distributed nature of data-centers. Its performance was based on letting switches explicitly communicate with end-hosts to let them know the available rate in their outputs, so that end-hosts could compute expected flow transmission times and packets could be scheduled accordingly. Although achieving great performance, it was quite a

complex protocol, which required a lot of synchronization between all parties as well as saving and exchanging a lot of information, both on switches and end-hosts.

For the scheduling algorithm to work in a distributed manner, flow information needed to be exchanged via explicit feedback in the packet headers. The main idea was that a scheduling header could be added in the transport layer of each data packet, in which the end-host would send information to the switches about the flow ready to be transmitted, and the switches would return information about the available rate at which the flow could be (at most) transmitted. When a new flow needed to start communicating, the sender would send a probe (a packet with no data), with the scheduling header specifying the new flow information (size and maximum rate that the sender could achieve). The switch would then specify the achievable rate in the header, which would be received by the sender through the acknowledgement. Every time a the receiver would get a packet, it just had to copy the scheduling header received in the acknowledgement. Switches in the path monitored the incoming traffic they had in each of their queues. They would explicitly ask the sender to transmit data at a specific rate or to pause transmissions by only modifying the scheduler header.

In 2013, **pFabric** [AYS+13], suggested that all research efforts towards the latency problem in data-centers had been trying to use rate control to reduce FCT for short-flows. Some solutions had tried to do it implicitly, by keeping queues near empty through adaptive congestion controls and ECN-based feedbacks (like DCTCP [AGM+10], D2TCP [VHV12], and HULL [AKE+12]). Although with those solutions, FCT was improved, they could not precisely determine the right flow rates. Moreover, keeping empty queues was challenging due to the bursty nature of traffic. The other solutions had tried to explicitly compute and assign rates from the network to each flow, to try to schedule them based on size or deadlines (like D^3 or PDQ). This last approach, although providing good performance, demanded too complex implementations. Believing that flow scheduling should be kept independent from rate control, pFabric proposed a simple transport design which managed to achieve near theoretically optimal FCTs.

- End-hosts were required to specify a number on the header of each packet, which would represent its priority (e.g. the flow's remaining size or the deadline). Priority would be set independently for each flow, with no need for coordination across flows or hosts.

- Switches would base their scheduling by just following the priority number on the header. Buffers would be given very little size and preemption would be considered: if a packet arrived with a full buffer, the packet with less priority would be dropped.

- Decoupled rate control was also simplified: flows would start transmitting at line-rate and would only decrease the rate in case of high and persistent packet loss detected. These two simple mechanisms would be sufficient for providing next-to-optimal performance.

Although the idea of using packet headers in order to specify scheduling priorities was interesting, and would be incorporated a bit later in the PIFO proposal, the rank definitions that pFabric suggested were based on SRPT. They assumed transport layer knowledge when performing scheduling, fundamental to compute remaining flow time deadlines through TCP acknowledgement information. Despite the outstanding results of this configuration, SRPT deadline definition was not expected to be an easy task, as will be investigated throughout this thesis.

While the algorithms summarized in this section just suppose a very brief collection of all the alternatives presented in latest years, the amount of efforts devoted by the research community on the field, evidenced the common believe that it was possible to find a single algorithm which could perform well in all types of situations, in all different case scenarios. After pFabric, however, one big question was put on the table. Is there actually a universal packet scheduling? Is there an algorithm which outperforms all the others in all situations? A final solution, that would solve all the challenges in packet queuing and scheduling field at once? First Anirudh Sivaraman et. al. in "No Silver Bullet" [SWSB13], and then Raddhika Mittal et. al. in "Universal Packet Scheduling" [MARS15], suggested that there is not a single technique achieving best results in all possible cases.

"No Silver Bullet" was first in stating the non-existence of a key algorithm to be used everywhere, anywhere, claiming that an election of the best scheduling candidate would strictly depend on what demands the applications running on top of the end-points would have. On a computer backup application requiring high throughput, the best way to process the packets would be completely different than the one for an interactive website demanding low page loading times. Following the same direction, "Universal Packet Scheduling" insisted that, although certain algorithms , when customized in particular manners, could be able to accurately replicate behaviors of other techniques, being the greatest Least Slack Time First (LSTF), there was not a universal packet scheduling outperforming the others in all type of situations on a strict sense. Both results enforced the need of changing the general research direction, and trying to work on making scheduling programmable. Instead of pouring efforts in the design of a single superstar algorithm, acting as a wildcard for all possible desired behaviors, the focus should be put on figuring out which technique, or combination of techniques (existing or new ones still to appear), would be the best ones to achieve the solicited specific requirements for every particular case scenario. Switches in the network would have to support a wide range of algorithms, with the flexibility enough to select the most adequate one at each moment to handle the diverse and continuously-changing application demands.

Recent advances in programmable switches, make us believe in the possibility of having chips supporting multiple schemes and being able to dynamically select the appropriate ones through the programs running on top of them. Making scheduling programmable is the natural next step of nowadays networks, but until reaching that point there is still a lot of progress to be done.

3 Our Algorithm

3.1 Introduction

While the latest changes in network programmability were quite promising and gave some fresh air to the networking sector, the scheduling field was not being part of this revolution. Anirudh Sivaraman et al. (MIT), thought that the reason why scheduling had not been made programmable in this new wave of arising programmable switches, was due to a lack of consensus in defining a proper abstraction, which could allow new schedulers to be defined and implemented under a common basis, serving as a reference for the research community to paddle in the same direction. In SIGCOMM '16, they proposed a new framework, aiming to make from scheduling programmability in switches a reality [SSA+16].

They argued that, on its base, any scheduling algorithm could be decoupled in two main steps. The first, was defining the time at which each packet should be transmitted, with relation to the other packets. This is what they called 'ranking process'. In it, each packet had to be given an appropriate rank value, identifying the ideal position that the packet should occupy in the queue before being finally transmitted. Ranks could be just seen as relative positions to be occupied by new packets, with respect to the ones already enqueued at the time of the scheduling decision. The second step, once the packet ordering had been defined, was to place the packet in the queue following this rank. For that, the key player was what they defined as the Push-In First-Out queue. A PIFO queue could be described as a "data structure that allowed packets to be pushed into arbitrary locations in the queue, but only dequeued from the head". If achieving such type of queue structure was possible, then the biggest majority of packet scheduling algorithms could be implemented by just designing the appropriate ranking arrangements among them.

With this new abstraction, scheduling in switches could be made completely programmable. Algorithm designers would only have to define a new ways of ranking packets according to the new requirements aimed to face. PIFO queues would do the rest, serving packets in the orders specified. As the PIFO mechanism could be shared by all algorithms, and the only variable component was the ranking definition, switches with PIFO implemented would allow scheduling programmability by just running new software in which new rankings were defined. In P4 switches, new P4 programs would be created, with scheduling algorithms tagging packets with the desired locations, so that every time a new packet arrived the intrinsic PIFO queue of the hardware could execute these policies by allocating the packets in the corresponding orders. While this proposed abstraction was really outstanding and will, for sure, suppose an inflection point in the scheduling history, Anirudh's work kept the focus on the design of this new type of queues in hardware, pushing the vendors and silicon foundries to start working towards this new paradigm. Knowing that ASIC design is really tough, and that hardware production is usually a multi year process, it is reasonable to think that some time will need to elapse until

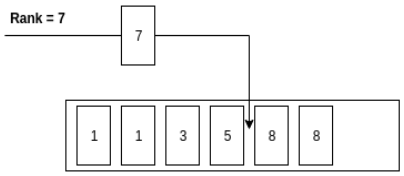


Figure 3.1: Push In First Out (PIFO) queue.

we can enjoy a definitive hardware-based PIFO implementation.

It is for this reason, that we asked ourselves the following question. Is it possible to implement an approximation of PIFO queue, with the tools supported in nowadays network equipment? If the answer is a yes, this means that we don't need to wait until new hardware is released to work with this new PIFO abstraction. And this means, that we can easily deploy new scheduling algorithms in an easy manner by just writing new tagging mechanisms, even in the switches deployed in current networks. If we can achieve an accurate approximation of PIFO with, for example, priority queues or FIFO queues, then theoretically we can do it in all kind of switches, not only Barefoot programmable ones. So giving answer to this question is undoubtedly of remarkable value.

3.2 On strict priority PIFO's

Let's define the working scenario as follows. Assume that we have a list of packets, in the header of which there is a field with their given ranking. This ranking defines the position these packets will need to occupy in the queue, and can measure a wide variety of parameters, depending on which objective the ranking-definition was designed for. Just for the sake of example (we will further analyze detailed algorithms for different performance objectives later on), in Start-Time Fair Queuing (STFQ), which is an approximation of the theoretical fluid model of WFQ, the ranking value would be the virtual start time, at which packet transmission should start[1]:

Algorithm 1 Start Time Fair Queuing Rank Example

```
1: f = flow(p);
2: if f in last_finish: then
3:     p.start = max(virtual_time, last_finish[f]);
4: else
5:     p.start = virtual_time;
6: end if
7: last_finish[f] = p.start + (p.length / f.weight);
8: p.rank = p.start;
```

STFQ computes the theoretical virtual start time at which a new coming packet should start its transmission. This time is selected as the virtual time at which the last packet from the same flow should finish transmitting, or the current virtual time in case the new packet's flow has no previous packets in the queue. As can be seen, in this case the packet ranking selected by the scheduling algorithm has the meaning of a virtual time, but another algorithm could select the rankings according to the packet length, with the rank meaning being bits, or just assigning

[1] Example extracted from PIFO original work [SSA+16].

simple weights. The importance is not what the ranking actually means, but how this ranking is assigned to different packets and which is their relation. At the end, PIFO queue will have to order coming packets based on this ranking without having any clue of where does this ranking come from and what it actually means, just as a mere ordering definition.

The ideal PIFO queue receives the packet with the assigned ranking, and places it in the position specified by the rank, pushing it in between of previously enqueued packets with higher and lower ranks respectively, without modifying the relative order of the packets already in the queue. Now the question is, how can we approximate this behavior with the type of queues we have in current switches? The majority of them support several FIFO queues, which can be configured (at most) to have different priorities, meaning that the queues with lower priority will not start draining packets until higher priority queues have been completely emptied. While a single FIFO queue does not provide us any opportunity to exploit, priority queues allow having some control about which packets will be scheduled first and which ones will have to spend more time in the queue until being finally served. We will take advantage of this control to try to approximate the behavior of the ideal PIFO queue.

Let's assume the following example, in which four consecutive packets are received, with rankings 2, 4, 1, and 10 (milliseconds if we continue with the STFQ algorithm, although we have already seen that the metric is not important). If we have a single FIFO queue, the first packet arriving (the one with ranking 2), will be enqueued in the first position; the second one in second position; the third in third position and the fourth in the last position of the queue. The output order we will obtain is the order of packet arrival. Instead, we would have liked our queue to dequeue the thirdly arriving packet (with ranking 1) in first position, then the firstly arriving packet (with ranking 2) in second position, then the secondly arriving packet (with ranking 4) in third position, and finally last arriving packet (with ranking 10) in last position. With just a single queue there is no way to achieve this result. But what happens now if we have four queues with four different priorities. Now intuition tells us that if we put the first packet in the second queue, leaving the first for the third packet, and placing the latest packets in the remaining lower priority queues, we will achieve the order that we want. Also intuitively, one can start to see that if we have a number of queues equal to the number of ranks, a perfect approximation can be achieved, as one should just assign each ranking to a queue, according to their priorities. But note that in order to do that, both with a limited number of queues or non-limited, knowledge of ranking of future coming packets is required. If a complete knowledge of the coming packets and their respective ranks is available, then scheduling the packets in priority queues in a strategic manner to leave room for higher priority ones that will come afterwards is quite straightforward. The problem comes when there is no knowledge about future packets (which is the case we have in networks), and some prediction needs to be done. In the same direction, if the range of rankings that you can expect can vary with the time (as application flows with strict requirements arise and vanish), the allocation of those rankings to the strict priority queues is not easy anymore. Following the example we had, with 4 queues we can achieve the required granularity for the 4 orders of ranking. But what if a new packet joins the scenario with a rank assignment of 3 (corresponding to a new application just started, for instance)? Now ranking number 3 will have to share queue with any of the previous ranking orders. And then we will not have complete granularity in the output. If rank 3 shares queue with rank 4, we will not be able to guarantee that all packets from rank 3 will depart before than the ones from rank 4, as the queue-sharing will provoke blocking situations among the two packet types. And the opposite situation is also bad. If with these 5 types of ranking, now arrivals of type 1 packets stop coming, the queue previously assigned to this type of packets

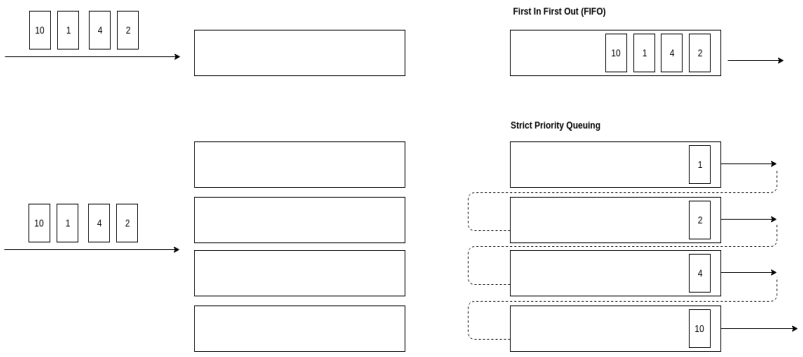


Figure 3.2: Strict Priority queues can be used to achieve finer ordering.

will become empty. If no update is performed in the packets distribution to the queues, this empty queue will suppose a loss of resource efficiency for ourselves. And that resource could be very useful, for instance, to give room to packets of type 3 and 4 to enjoy a private queue each of them and achieve this way the finer granularity we were desiring since the beginning.

This toy example allows us to illustrate two interesting remarks. The first one is that it is possible to implement an approximation of the ideal PIFO behavior by just using priority queues. With the insight that the bigger the number of queues, the finer the granularity we can achieve (the better the approximation). And the second is that finding the right way to distribute packets with different ranks among the different queues available is not an easy problem. A good solution would be the one that adapts to possible changes in the ranking definitions of the packets, intrinsic from the dynamic nature of the network, as well as trying to maximize resource efficiency at every moment.

3.3 What is a good strategy and what makes it good?

If we want to design a proper algorithm, we first need to be able to define what distinguishes a good allocation strategy from a bad one. We have multiple ways of allocating a limited number of packets to a limited number of queues. How can we compare two different strategies? How can we know which one is better? And more importantly, how much better is one from the other? If we are able to define what makes an allocation strategy a good or a bad strategy, we will be able to find mechanisms to continuously achieve good strategies for our solution.

One way of evaluating if the allocation strategy is good or bad is to analyze the sequence of packets derived at the output. At the end, the final objective of a PIFO queue is not other than ordering packets. In a dynamic way, yes, as packets come and leave. But the objective is to send them in order of ranking. The traditional mechanism to define whether a sequence is properly ordered, or to quantify how properly ordered a sequence is, is the called inversion number.

We can define an inversion in a sequence of elements, as a pair of consecutive elements which appear out of order according to a sorting mechanism clearly specified. In a formal manner: Let A(1), A(2), ... , A(n) be a sequence of n numbers. If A(i)>A(j) and i<j, then the pair (i,j) is referred as an inversion of A. The inversion number of a sequence is a common measure of its sortedness. The inversion number can be defined as just the number of inversions that can be found in a sequence. Inv(A) = # {(A(i), A(j)) | i<j and A(i)>A(j)}.

As an example, the sequence {9, 5, 7, 6} has an inversion number of 4, with inversions (9-5), (9-7), (9-6) and (7-6). Although the inversion number has been broadly used in a variety of fields, in our case it is important to distinguish inversions between close rankings in a sequence, and those rankings more away from each other. For instance, it is not the same achieving an output sequence in which a packet of rank 2 has been enqueued previous to a packet of rank one, than if the inversion occurs between packets of rank 1 and 15. In order to measure this case distinction, we will define a new metric, which we will call weighted inversion number, in which each inversion will not add a value of one to the inversion number, but the difference between the inverted elements respective ranks will be added instead. In the sequence example {9, 5, 7, 6}, the inversion (9-5) will have a weight of 4, the inversion (9-7) will have a weight of 2, the inversion (9-6) will have a weight of 3 and the inversion (7-6) will have a weight of one, adding up to a final weight inversion number of 10. These two metrics will allow us to determine if a packet allocation strategy is good or not. We will always look for low inversion (and weighted inversion) numbers. Those numbers will be the ones expressing if, whatever it was the sequence in which the packets entered the scheduler, the scheduler was able to correctly order them following the rankings specified, and the final output sequence has a good level of sortedness.

Now that we know how to evaluate an allocation strategy and we have the metrics to quantify how good it is, which allocation strategies can we follow? In a scenario in which we receive k consecutive packets and we have n available queues (suppose of unlimited length), we have n^k different allocation possibilities. Obviously, determining if an allocation has been good or bad can only be done a posteriori, when all the packets have been allocated and we can measure how good is the output sequence obtained as result. We can never guess how good is the allocation of a certain packet to a specific queue until we can see which packets will come next and how that choice will affect. Although finding all the possible allocations is not useful in a practical sense, analyzing the different behaviors can bring us an idea of which kind of problems will need to be faced when trying to design a proper solution algorithm.

For this reason we have created the small program that we will describe in the following lines. It is a small Java-based simulator, that allows the user to compute all the possible allocation strategies that one can follow in order to place a certain amount of packets (to be received consecutively) into a number of queues. The software not only gives all the different strategies that can be performed, but also analyzes them in terms of both, the inversion number and the weighted inversion number. Its functioning is quite simple. Just imagine a scenario with two different queues, and an incoming sequence of packets with ranks 3, 2 and 1 in this order of arrival. When the first packet arrives (the one with ranking 3), we have two allocation possibilities, placing the packet in either the first or the second queue. When a second packet arrives, we can also put it in the first or the second queue. Combining the first and the second packets allocation, we have a range of four different possibilities. The same applies for the third packet, giving now room for 2^3 possible allocations, and again for all subsequent packets. The software just implements this iteration in the form of a tree, in which each time a new packet comes, the current distribution of packets in the queue is replicated multiple times according to the number of positions into which the new packet can be placed. Figure 3.3 exemplifies this behavior with the illustrated example.

From all the possible packet allocations, we have two candidates which outperform the others (i.e. the ones with inversion number of 1). The key of their success is that they place the packet with ranking 3 in a lower priority queue, allowing following packets with lower ranking to be allocated in higher priority queues and to be transmitted in first place. This shows again, that

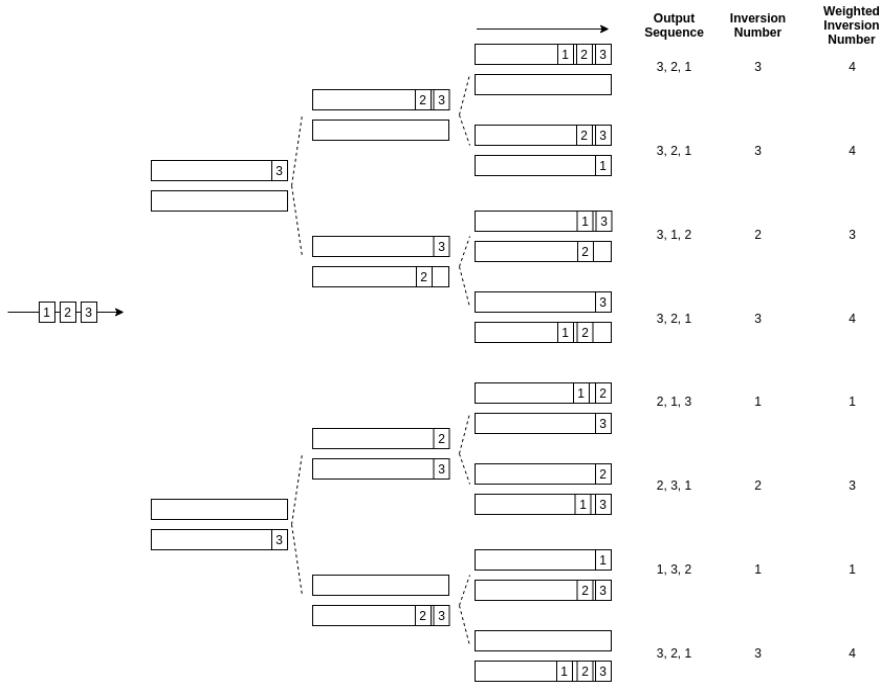


Figure 3.3: Possible packet allocations for a number of available queues.

Algorithm 2 Pseudo-code to compute all possible allocations for a given number of packets and strict priority queues.

```
1: /*Random packet sequence generation*/
2: for (j<num_packets) do
3:     Packet p = new Packet(rand.nextInt(max_rank));
4:     packets.add(p);
5: end for
6: /*Creation of tree root*/
7: Node root = new Node(num_queues);
8: /*When a new packet arrives, leaves are forked with children inheriting existing queues,
   and new packet possibilities are computed*/
9: for (i<packets.size()) do
10:    forkLeaves(root);
11:    Packet pk = packets.get(i);
12:    addPacketLeaves(root, pk);
13: end for
14: /*Print result*/
15: traverseTree(root);
```

the main challenge we will have to deal with is predicting future packet ranks. As knowing which packets will come in the future is realistically not possible, what we will do instead is try to minimize the possibility of collision, where a collision occurs when a lower rank packet has to be placed after a higher rank one, generating an inversion in the output sequence.

Another interesting perspective to the problem is trying to answer the following question. How many queues would we need, to avoid collision for a specific sequence of packets? As the origin of collisions is the fact that most of the times we can not dedicate a queue for each specific rank level, it makes sense to analyze how many queues are needed for a given sequence to avoid congestion. The direct answer is the number of ranks contained in the sequence. Thats the PIFO ideal case. But, can we specify a more tighter bound? For the sequence {3, 2, 1} we will need 3 different queues to achieve a perfect ordering, and thats the best we can do. But for the sequence {3, 1, 2}, with the same ranking levels, only two queues are needed to achieve a perfect ordering. In the same direction, {1, 4, 2, 4, 3}, also needs just two queues, while the ranking levels are four. Which is the rationale behind that?

It is clear that by having a specific queue for each rank level, we can prevent our system from collisions. The key point is trying to analyze if different rank levels can share the same queues without this creating collisions. By defining the maximum (and minimum) position of a rank level in a sequence as the position held in the sequence, following arrival order, of the last (and first) packet pertaining to the rank, we can state that consecutive rank levels will be able to share the same queue without collision occurring if and only if, the minimum position on the sequence of the higher rank is higher than the maximum position on the sequence of the lower rank. By analyzing the maximum and minimum positions of each rank in the sequence, we can compute the number of queues required in the strict priority scenario to achieve a perfect ordering at the output, equivalent to the ideal PIFO one.

Knowing how many queues do we need to achieve a perfect sorting can provide us some interesting insight. With this knowledge we can compute the number of permutations, for a given set of packets (now ignoring the rankings they have attached) that we are able to achieve with a given algorithm. One could consider each algorithm as a system providing a specific output for a given input sequence, If a given input σ can produce a particular output τ, then (σ, τ) it is said to be an allowable pair. Computing the amount of allowable pairs that a system is able to provide can give insight on how good or bad an algorithm can be in terms of sorting. With PIFO, for example, n! different sequences can be achieved at the output, given that the input is a sequence of n packets. For n=1, the input set is {1}, and therefore there is only one possible permutation. For n=2, the set is {1,2} and accepts two different permutations: {1,2} and {2,1}. For n=3, the set is {1,2,3} and the possible permutations are {1,2,3}, {1,3,2}, {2,1,3}, {3,1,2}, {2,3,1} and {3,2,1}, which are 3! in total. As PIFO is able to perform perfect ordering of packets entering the queue, all the different permutations at the output are achievable from the original set {1,2,3}. Therefore, for n=3, PIFO achieves 6 allowable pairs. With strict priority queuing instead, depending on the number of queues available, some outputs will not be achievable. For instance, the permutation {3,2,1} is not achievable with a strict priority scheme of 3 queues. The pair {1,2,3}, {3,2,1} is not an allowable pair for the strict priority scheme under 3 queues scenario. We have computed the number of allowable pairs when the input has n packets. The resulting graph can be observed in Figure 3.4, and clearly showcases that the bigger the number of queues available in the strict priority algorithm, the better the PIFO approximation it can be. This gives us confidence to think that it is possible to approximate PIFO with strict priority. Now the next objective will be designing an algorithm that correctly decides the best permutation to compute according to the input sequence and its requirements.

Algorithm 3 Number of queues needed to achieve perfect PIFO.

```
1: ArrayList<Packet> packets;
2: ArrayList<Tuple> tuples;
3: /*Compute max and min position of the rank in the sequence*/
4: for (int i=0; i<packets.size(); i++) do
5:     if (!t.contains(tuples, packets.get(i).getRank())) then
6:         Tuple tup = new Tuple(packets.get(i).getRank());
7:         tup.setMaxpos(i);
8:         tup.setMinpos(i);
9:         tuples.add(tup);
10:    else
11:        t.getTuple(tuples, packets.get(i).getRank()).setMaxpos(i);
12:    end if
13: end for
14: Collections.sort(tuples);
15: /*Compute number of queues needed to achieve perfect PIFO according to max-min
    positions*/
16: int numqueues = 1;
17: for (int i=1; i<tuples.size(); i++) do
18:    if (tuples.get(i).getMinpos() < tuples.get(i-1).getMaxpos()) then
19:        numqueues = numqueues + 1;
20:    end if
21: end for
22: System.out.println("Result: " + numqueues + " queues are needed ");
```

3.4 Design of a dynamic allocation algorithm

Until this point, we have only analyzed the cases which we can find when allocating packets under 'a posteriori' criteria, checking the conflicts and scenarios once all the packets have been scheduled by analyzing the resulting output sequences. Now we will change the perspective of the problem towards a more dynamic view. This means that instead of knowing the full input sequence of packets, we will try to think in the same way the scheduler will have to, at a per packet basis. The only thing the scheduler knows is that it has received a packet with a rank. At each allocation decision, the scheduling does not have any information about future packet ranks, nor statistical distributions.

We have seen that when a wider number of queues than the number of ranks is available, perfect PIFO can be easily achieved. Now, what happens when thas scenario is not possible? Which would be the ideal algorithm, under the assumption of a finite number of queues, when the number of queues is way smaller than number of possible ranks, each queue with infinite buffer length, and only knowledge of each packet rank under arrival (meaning that we don't have any information about future packet ranks, nor statistical distributions)? One could easily think at first that the ideal algorithm would be the one allowing to modify the packets which have been already enqueued, so that in case one of the packets in queue blocks a new one which is coming, the enqueued packet could be switched to another queue and give room to the coming one. Obviously again this is not realistic, but helps us to identify once more what we will look for in our solution.

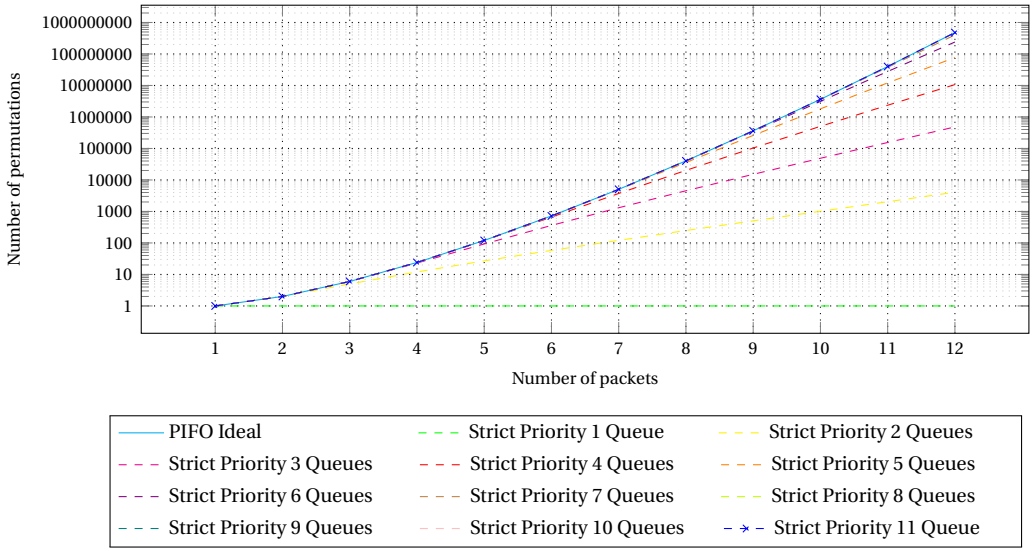


Figure 3.4: Number of permutations that each system is able to sort.

As predicting traffic is not always possible and switching enqueued packets throughout queues can not be done, our designed solution is based on the paradigm of trying to minimize block probabilities. Several strategies can be followed when trying to deal with blocking events minimization. Treating all packets equally in terms of importance and just placing them in a way that the maximum number of packets can be allocated without blocking. That would be one design strategy. The other would be giving more relevance to packets with lower ranking, for instance, so that when blocking occurs, the ones receiving the worst effects are the ones with higher rankings. More complex ideas would include trying to compute, store, predict and use traffic distributions in terms of ranking, to adapt the queuing decisions following these patterns. However, this would require keeping state of the traffic statistics, and describing a mechanism to dynamically update them to the changes in the network. Although saving state is possible in current programmable switches, complex implementations are not recommendable with the state of the art technology. Those complex algorithms would probably offer a better performance. Note that we have designed this algorithm to be as simple as possible, with the eye put in making it possible to be implemented in current versions of P4 switches. As they don't allow loops to take place, and the number of states which can be saved in a practical manner is not huge, we had a threshold between performance and feasibility in terms of implementation. Our goal was not to achieve a mathematical optimum in terms of allocation results, but to achieve an implementable version in the constrained versions of programmable switches we can have nowadays.

3.5 Proposed solution and why it works

I. For each arriving packet, the ranking specified on its header is obtained. Queues are tracked bottom-up (from less priority to higher priority), observing which is the ranking level contained in the last enqueued packet from each queue. If the last packet ranking is minor or equal than current packet's one, the packet is enqueued. Otherwise, the next queue is analyzed in the same manner and the procedure is repeated until reaching the last queue (highest priority). Packets reaching last queue will be enqueued whether their ranking is higher or lower than the one defining the queue.

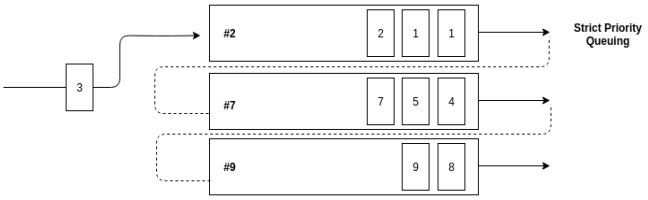


Figure 3.5: Strict Priority PIFO (SP-PIFO).

a) Ranking of new packet is obtained: 3.

b) Ranking of last packet in third queue is obtained: 9.

c) Is 9 <= than 3?

   No: Jump to next queue.

d) Ranking of last packet in second queue is obtained: 7.

e) Is 7 <= than 3?

   No: Jump to next queue.

   As next queue is the first one, packet will be enqueued in the first queue. In this case 2 is smaller than 3, so there is no blocking. But even if there was blocking, packet would still be enqueued in that queue.

II. After having understood the basic procedure now let's move to the adaptive part, which is detecting blocking and acting to prevent repetition. For this, we need to define the concept of queue level. In the basic procedure of the algorithm, the queue level is the ranking of the packet which was enqueued last. In Figure 3.5, the highest priority queue has a level of 2 when the new packet arrives, mid-priority queue has a level of 7 and low-priority queue has a level of 9. When the new packet is enqueued, high priority queue updates its level to 3, following the rank of the newly enqueued packet. This queue level is the fundamental concept when trying to control blockings in the network and react accordingly.

   The key point here is to understand why blocking is happening in our algorithm and what does it mean. Then, what can we do to react and prevent it from possibly happening in future events. When starting to run the algorithm, all queues are empty, and initialized with a level of zero. Then the basic procedure takes place, with packets filling the queues and moving to the top as packets with different ranking levels come. When the queue filling reaches the first queue, there is no more space to keep expanding, and therefore ranking granularity has reached its possible end. It is in this moment when blocking can start occurring in the highest-priority queue. Blocking can be easily detected by comparing the rank of the new packet with the level of the queue. If the rank is lower than the level, blocking is going to take place. The fact of blocking happening informs us about the need for higher level of granularity in the top queue. It occurs when high rank packets are sharing the same queue as low rank packets, which would like to enjoy their assigned priority. Blocking therefore shows a problem of queue space granularity for the different rankings. And while not always happens, some times is possible to reach the desired level of granularity by re-designing the distribution of packets throughout the other queues, trying to squeeze the maximum efficiency in the distribution of rankings among available resources. To move some packet assignments to the lower queues,

queue levels of smaller priority queues need to be decreased. This way, as packets start the procedure by checking the queues below, if they experience lower levels, there will be a higher probability that they will finish enqueued in those layers. And, as it is not only important to detect if there is blocking to react, but also to know if the blocking is big (wide difference from new rank to level), we will use the weighted inversion index to measure the blocking and react accordingly. To verify the proper performance of the designed algorithm, we have also developed a Java version to compare the obtained distributions with the theoretical ones, and see how close we are from the ideal results.

As can be seen in the snippet below, the only required state to be saved in the switch is a variable queue level for each of the available priority queues. This state will be compared with the packet rank to determine the right allocation position, and after enqueuing the packet, in case blocking occurred, all the below queues will have their levels decreased an amount equivalent to the weighted inversion number of the blocking, to re-distribute weights among the queues and prevent possible blockings in future allocation decisions.

Algorithm 4 Scheduling of each packet in the corresponding queue

```
1:  for (int i=0; i<packets.size(); i++) do
2:      for (int q=queueList.size()-1; q>=0; q- -) do
3:          if ((queueList.get(q).getLevel()<= packets.get(i). getRank()) || (q==0)) then
4:              int WIN = queueList.get(q). addPacket_and_getWIN(packets.get(i));
5:              if (WIN > 0) then
6:                  for (int w=queueList.size()-1; w>=q; w- -) do
7:                      queueList.get(w).setLevel(queueList.get(w). getLevel()-WIN);
8:                  end for
9:              end if
10:             break;
11:         end if
12:     end for
13: end for
```

Example of execution 3.5.1. For a scenario with 70 packets, 5 queues and maximum rank set to 9:

Random sequence of packets generated is:
<- - - - {2}{2}{3}{6}{5}{1}{1}{5}{2}{3}{7}{7}{7}{3}{6}{5}{8}{2}{3}{8} - - - -
Packets in queue 0: {1}{1}{2}{3}{3}{5}{2}
Packets in queue 1: {5}{5}{6}{3}
Packets in queue 2: {2}{2}{3}{6}{7}{7}{7}{8}{8}

Output sequence when draining is performed after all allocations:
{1}{1}{2}{3}{3}{5}{2}{5}{5}{6}{3}{2}{2}{3}{6}{7}{7}{7}{8}{8}
Inversion number: 25
Weighted inversion number: 55

3.6 Pros, cons and discussions

Self-learning and adaptiveness: As mentioned previously, one of the most important aspects of the algorithm is reacting to changes in the network. PIFO should be able to order any set of

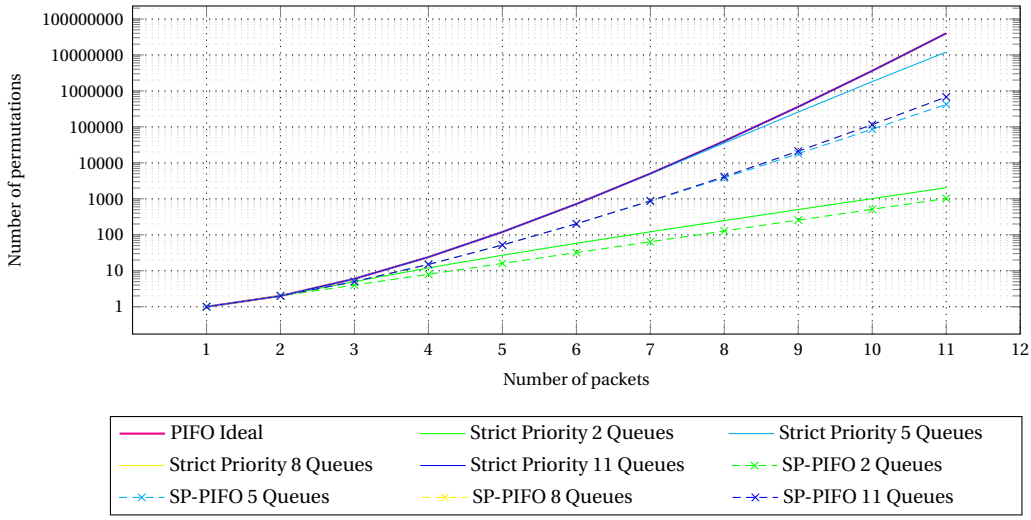


Figure 3.6: Number of permutations that SP-PIFO is able to sort.

ranks, allowing new arrangements to be developed and deployed on the fly. Some of them, like pFabric, will be implementable directly from end-hosts, by just introducing a new header to the packets or modifying existing fields. Others, will require coordination of switches along the path to specify the correct order, taking advantage of local information like queue depths, flow characteristics and output link capacities. There will be even some, trying to mix both praxis, with a first rank initialization at the end-host and ranking modifications throughout the path. In all cases, PIFO needs to be capable of adapting to the new ranks specified, both if they are big or small, and providing transparent progression as much as possible from the old sets to the new ones. In our proposed solution, thanks to the designed blocking reaction and the use of weighted inversion index, this changes can be detected and reacted in a fast rate. Moreover, as blocking is centralized at the highest priority queue, which is the one being drained in first place, their effect will be minimized, resulting on a seamless impact in the end-user quality of experience.

Fast start for fast convergence: One of the main problems in the proposed algorithm is that, even when dealing with uniform traffic, it takes some time until converging to the most accurate queue-level distribution. Any alternative to better allocate queue levels from the very beginning, overcoming the transient time of stabilization, would require previous knowledge of the upcoming traffic to be received. As can be seen in the Example 3.5.1, the majority of inversions take place at the beginning of the execution, when queues are filling up. Although when running the algorithm in real time and draining the first packets while the queues are filling (instead of computing the whole allocation and draining at the end), this problem will vanish, we have liked to think about a possible rapid-convergence alternative. The objective is to prevent blocking from moment zero. When first packet comes, it will be placed in the middle queue. In case of working with a pair number of queues, just select one of the middle two. When the second packet comes, in case its rank is higher than the existing packet one, it will be placed in the middle queue of the queues above the one which is occupied by the first packet. In case the new rank is smaller, exactly the same but among the empty queues below. The idea is to keep the maximum space among filled queues, to give room for possible new ranks of coming packets. The price that is paid in exchange is complexity. This fast-start would need to be combined with the basic algorithm after the transient has elapsed. A highly-related discussion will be brought up in Section 4.3.2, in which we will see how resetting the queue

levels to zero after they have finished draining all packets does not help in the performance because of the bursty nature of the traffic. Instead, keeping record of queue levels is a better option, as with high probability, the traffic distribution will not change from burst to burst, and we will prevent the scheduler from having to go through the transient queue level adaptation period again.

Traffic knowledge in advance: Note that in general, any type of information that one might have about incoming traffic characteristics is positive, and can be used to achieve higher levels of personalization, and therefore accuracy, in the deployed solution. For instance, in an example with four queues, and a uniform traffic distribution among packets of eight different types of traffic, the best allocation would be just assigning in order two consecutive ranks to each of the queues. It is interesting to think about the advantages of knowing the traffic statistics in advance. Even just the maximum and minimum of the ranking levels that can be received, can suppose insights enough to improve the algorithm performance, making it focus on the particular scenario. If traffic distributions are known in advance, again, dedicated algorithms will outperform any adaptive one aiming to serve all types of scenarios. Working with traffic distributions, however, would require broad saving state capabilities, making the complexity grow if a fine granularity is willing to be achieved. In Section 4.3.3, an experiment proving how traffic knowledge can be used to improve the final performance will be showcased. In that particular case, our algorithm provides close to the optimal results, despite the intrinsic losses due to its transient adaptation time.

Preventing starvation: Probably the biggest drawback we can find when working with strict priorities is queue starvation. When high priority queues happen to be continuously full as low-rank packets are received at a high rate manner, it can happen that lower queues can never start draining their packets, making them spend high amounts of time in queue, waiting for the high priority queues to finish their transmissions. When that happens, what should we do? Is that avoidable from just the ranking definition algorithm? Is this really a task for the PIFO queue? Our position in this case is first stating that starvation is a characteristic not only coming from the use of strict priority queues as a basis, but intrinsic from the PIFO definition. Whenever lower ranked packets arise, they will have preference over higher rank ones, making the last ones starve no matter how PIFO is implemented. It is true that in the case of mono-atomic rank increase, which was already underlined by the original PIFO authors for being the most characteristic one in the majority of schedulers, this starvation can be fought with the help of a virtual clock taking count of the current round and giving priority to packets having arrived before in the queues. But even in those cases, we would suggest playing with the queue rate definition capacity that P4, and most of the switches, provide. By correctly configuring the queue rates, in a manner that the highest priority queue does not occupy all the output bandwidth share, lower priority queues will still be able to drain some packets preventing cases of complete starvation. It is important to keep queues small sized and empty for the correct behavior of the algorithm.

Queue depths, dropping and congestion control: Until this point, a single scenario has been analyzed in reference to the queue depth: the one in which all queues have unlimited queue length. However, in reality the buffer size is another variable to take into consideration. A whole field of research is focused on how to correctly design and configure queues, not only in terms of depth but also concerning drops and congestion prevention. Buffer management, as it is called, aims to achieve good queues, defined as those which allow temporary storage for incoming packets when the arrival of packets received exceeds the capacity of the egress link. As perfectly explained in [NJ12], good queues need to be designed for accommodating

the bursty nature of traffic and maintaining high levels of link capacity utilization. Bad queues, on the other side, are those which can never be drained and instead of dynamically filling up and draining in a natural manner, they just serve as a stopper for every packet that arrives. We will not focus on this well known buffer-bloat problem, but it will be interesting to think about which levels of congestion will be the appropriate ones to validate the performance of our schedulers, and how to generate good queues when implementing our simulations. Although we will not cover it throughout the thesis, our solution does not exempt congestion control and dropping management to take place. Algorithms like ECN and RED, could be also supported with the PIFO queues, and would be interesting to have implemented in order to achieve well-behaving queues and maximize performance results. In reference to dropping policies, we have only worked with the traditional tail-drop regime, in which each priority queue has its own depth and packets received later are just dropped. When the queues buffer-bloat, high rank packets will not only starve the lowest priority queues, but after the queue is filled, will start to be dropped. Starvation and dropping are intrinsic from the PIFO idea, whenever a pre-emptive approach is looked after, in which low rank packets will always have priority against low rank ones, even they arrive later to the queue. Any PIFO implementation will suffer, by nature, the starvation problem, and consecutive dropping problem no matter which queuing structure is being built on (even on hardware). This makes us think on the possibility of linking up ranking definition with dropping management. For instance, adjusting rank settings according to the state of the queues, so that dropping probabilities, or the effects on application traffic can be minimized. But then the question arises, is it really better to place for example the packet in a lower rank queue to avoid dropping or it is better to just drop it, so that the end-host can react to the congestion with a better retransmission. This type of dilemmas are the ones that, at the end, predictable networks will need to face.

Efficient use of resources: An important strength of our algorithm is that responds efficiently to the monotonically increase of packet ranks within a flow. One of the major PIFO observations was that in most scheduling algorithms, ranks used to increase within a flow. Packets arriving later inside a same flow were allocated higher ranks than their predecessors. This can be very easily seen for example in time-based scheduling like virtual start time in STFQ. The virtual start of future packets will be obviously bigger than the one in previous packets. This intrinsic nature of the ranks is the basis of its behavior, where packets in increasing level of ranks can just be consequently allocated to the same queue, making an efficient use of the existing resources. This characteristic also serves for guaranteeing that packets will be prioritized in transmission order, preventing packet reordering at reception, and also reduces the number of elements that need to be sorted by the scheduler.

A clear case for reinforcement learning: Finally, and just as a brainteaser, as it is out of the scope of this thesis, it is interesting to look at the PIFO problem from the machine learning point of view. It is compelling trying to know what would happen if we could let a neural network solve the problem for us, training the system with traffic arrivals and letting the weights be adapted according to the performance obtained after each given allocation. At the end, it can be seen as a chess game, in which each decision derives into a direct result, measurable, as a ground truth logic, to make the system learn and obtain the best allocation for the particular traffic statistical model. Of course, thinking of using machine learning for scheduling in real time would be too ambitious in terms of complexity and cost for what it is, but who knows if good strategies could be found off-line, to be applied afterwards for determined traffic models or statistical patterns.

3.7 Wrap-up

- What we have learned:
  - It is possible to approximate PIFO with strict priority queues.
  - The higher the number of queues, the better the approximation.
  - If the number of queues is equal to the number of rank levels, a perfect approximation can be reached.
  - Sometimes, depending on the particular input sequence, a tighter bound can be found in the number of queues needed for a perfect approximation.
    - We have defined the way this bound can be computed, and have implemented it on software.
  - We have described a new metric to evaluate and quantify the sortedness of a sequence: inversion number and weighted inversion number.
  - There are n^k different possible allocation strategies to distribute n packets in k different queues.
    - We have implemented an exhaustive research program to compute and analyze them in terms of the two inversion metrics.

- Finding the right way to distribute packets in a finite and small number of queues is not an easy task:
  - Players:
    - Number of packets (in posteriori scenario, but equivalent to difference between arrival and departure rate).
    - Set of possible ranks.
    - Number of queues.
  - Challenges:
    - Adapting to possible changes in ranking definition or variations in traffic arrivals.
    - Making efficient use of available resources (queues) to achieve the highest level of granularity at each moment.

- Approach followed:
  - Minimize the number of collisions among all different rank levels, when treated equally.

- First assumption:
  - No information about future packet ranks, nor statistical distribution.
  - Once a packet is enqueued, can not be switched from queue anymore.

  - Proposed solution:

        * All queues initialized at level zero.
        * Queue-levels are checked bottom-up. If queue level <= rank, packet enqueued. If analysis reaches top queue, packet enqueued.
        * Blocking is centralized at top queue.
        * To prevent future blocking, spreading down granularity a weighted inversion number amount.
    - Advantages:
        * Only need to save a single state per queue.
        * Simple implementation.
        * Adaptive to changes.
        * Blocking happens in highest-priority queue, minimizing effects in QoE.
    - Drawbacks:
        * Transient until some stability is reached.
            · Possible alternative: Fast start, aiming to use efficiently queues from beginning, try leaving maximum space between non-empty queues. More computational complexity required.
    - Open questions:
        * Is there a way to find the mathematical optimum without need for computing all the possible allocations, to compare SP-PIFO performance to the PIFO theoretical one?
        * Should we take care of starvation of low priority queues? Or should it be a task of transport control protocol via retransmissions with lower rank? Remember that the rank can be specified at the end-host and modified by the path switches.
        * In the same direction, should we play with queue size? Or is dropping and buffer management a task of the scheduling algorithm, not PIFO itself?
        * Should we treat better some ranks among others in terms of queue allocation?

- Second assumption:
    - Predicting traffic is possible. Compute, store, predict and/or use traffic distribution statistics in terms of ranking can give several advantages. Main question is if traffic statistics are stable enough to be worth the amount of state-saving needed to implement such approach. Is this implementable in current hardware?
        * A simpler approach is to model certain traffic scenarios off-line, and try to take advantage of this knowledge a posteriori to efficiently allocate the resources.
        * Learning algorithms will be completely necessary when trying to mix different performance objectives under the same switching equipment.

4 Experiments and Results

4.1 Introduction

After having analyzed the proposed algorithm in a theoretical manner, in this section we will tackle its practical implications. The final objective of developing an approximation for PIFO queuing behavior, is to make possible the implementation of more complex scheduling algorithms than the ones we can find intrinsically in nowadays switches. The continuously increasing demands for quality of service, evidences the need of complex algorithms defining the way packets should be treated to achieve certain objectives. In the previous chapter we have defined a way to approximate the PIFO behavior by using strict priority queues, a basic queuing style that we can find in all the switches deployed in current networks. In this chapter, we will focus on the other crucial point in the PIFO abstraction: setting ranks. We aim to verify if the designed PIFO approximation can actually be useful in practical implementations of complex scheduling strategies.

A variety of scheduling algorithms have been presented since the appearance of first networks. All of them try to face different performance objectives: helping flows achieve fair bandwidth shares, minimizing packet flow completion times, reducing one-way delay tail latencies or even aiming to find a balance among combinations of them. In this chapter, we will select the most representative ones, and we will discuss how they could be adapted or translated to work under the PIFO abstraction. After implementing them in P4, we will verify if, with our solution and the appropriate rank allocation design, we are able to approximate the behavior that those algorithms were developed to provide, with a certain level of accuracy. It is very important to remark that our evaluation is completely focused on translating results that up to now have only been achievable in simulations, dedicated hardware or research testbeds, to existing networks and equipment. Following this purpose, we have decided not to use any type of software simulator like ns-2 nor queuing modifications on Linux traffic control. Software tools allow researchers to extend the functionality of real hardware equipment further than what they can really provide in real life. Although sometimes that enforces research, as allows the community to test possible future enhancements and protocol ideas, this extra flexibility and programmability that software provides, could be easily perceived as cheating, bringing us away from our initial mission. Instead, the only tools we will use are Mininet [LHM10], a tool allowing the definition of realistic virtual network topologies on software to make testing easier, and the simple-switch definitions of P4 programming language, which can be run in the recently presented new generation of programmable switches. For the ease of implementation, we will focus the majority of experiments on the software virtualizations. However, in Section 4.6, we will showcase how the algorithm can perfectly be translated to hardware equipment. Not only we will provide the complete PIFO implementation for the most recent programmable silicon chip in the world, but we will also explain the main limitations that jumping to the physical world can suppose.

4.2 Setting ranks

Defining the way in which packet ranks have to be set in order to follow a specific scheduling algorithm is not always straightforward. The main reason behind is that algorithms were not initially designed for running on top of a PIFO abstraction. When they were developed, nobody was thinking about algorithms as a particular rank distribution. Although the first scheduling proposals were quite simple from an implementation point of view, and their translation to the rank paradigm is rather simple, as the scheduling demands of fairness, delay and bandwidth started to diversify, the presented solutions also grew in complexity. Adapting those algorithms to the PIFO paradigm requires a deeper amount of effort and time. In the lines below, we will suggest some mechanisms to define ranks in programmable schedulers, which will allow us to achieve most of the traditional and widely-used functionalities. Those definitions will be followed by experimental simulations, which will help in remarking the advantages of each proposal, as well as their limitations and improvement opportunities.

The analysis will be divided in two main blocks. The first one will cover scheduling algorithms that can provide benefits to the network even if they are only implemented in a single switch. For testing those algorithms, a basic single-hop architecture will be used. The second block will cover those algorithms which work in a distributed manner along the path, tweaking packets specifically at every hop, to provide end-to-end results. A more complex topology will be used for those techniques.

In both cases, the network equipment will be based on the P4 simple-switch target, a virtual implementation of a programmable switch, which allows different specifically designed P4 programs to be loaded and run on top. By generating traffic from origin-hosts and monitoring the different metrics of interest from Python programs running on destination-hosts, the different performance objectives will be validated. It is important to note that P4 behavioral-model, the framework that allows us to implement software on top of the switch, has not been designed for high-performance testing, but just to validate ideas and recreate proof of concepts. This means, that we do not expect to achieve state-of-the art performance results with these simulations, but to just verify if it makes sense to use a PIFO approximation for scheduling and if it is true that different objectives can be decently replicated.

4.3 Single-hop algorithms

4.3.1 First Come - First Served

Although P4 queues are FIFO by default and therefore a PIFO-based FCFS is not useful in practice, it is worth devoting some time to understand how this behavior could be achieved, as it is an easy-to-understand and very illustrative example of how setting ranks in PIFO can adapt the scheduling decisions to everyone needs. Moreover, for future scenarios in which PIFO has been established as the by-default queue in programmable switches and the open side is to just set the ranks of the packets on the go, knowing how to achieve FCFS behavior under a PIFO queue can also deserve some relevance. Under the PIFO philosophy, FCFS can be achieved in different manners. The most intuitive one is setting arriving packet's ranks to the time stamp of the moment when they enter the switch (this is, at the ingress). Simple-switch target already provides us with a time-stamp at the ingress, which we can directly use for setting the rank:

```
meta.rank = (bit<32>)standard_metadata.ingress_global_timestamp;
```

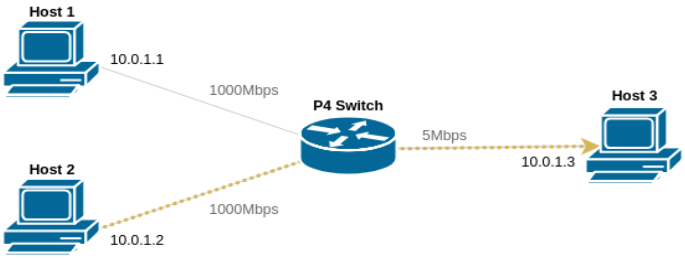


Figure 4.1: Single-hop algorithms topology.

To evaluate that the proposed rank definition behaves as expected, the order of departure from the switch is checked and compared with the arrival order. This can be done by tracking order of arriving packets at reception, and comparing it with sequence number of received packets. For the traffic generation, *iperf* is used, with a 5.5Mbps UDP configuration and a duration of 10 seconds. Figure 4.2 (top) shows how, under this definition, packets are received in order at the end-host. This rank definition is not the only way to achieve a FCFS behavior. Another option is to set the same rank to all packets. By doing so, all the packets will be directed to the same queue, without ones being prioritized above others. The queues are FIFO by nature, so the order in which packets arrive will define the order in which they depart. In Figure 4.2 (bottom), the two techniques are compared to the P4 intrinsic FIFO (this is, when no PIFO is implemented) in terms of flow completion times when 15 TCP flows with different sizes, ranging from 4000KBytes to 60000KBytes are simultaneously transmitted. It can be clearly seen that the three alternatives achieve equivalent results. The main difference of both PIFO implementations with P4 intrinsic FIFO behavior is the amount of operations that are performed in the PIFO pipeline. In particular, as only the first queue is used, that time is the required to read and write one single register in the switch per packet. Although that may sound a high price to pay, programmable switches are expected to execute those programs at line rate, meaning that in real hardware implementations, both the intrinsic FIFO and the FCFS PIFO abstractions would result in the same level of performance.

An interesting remark of FCFS algorithm is that it represents the perfect example of the mono-atomic rank increment nature. As all consecutive packets will have progressively higher arrival times, their allocated ranks will also follow an increasing trend. Therefore, following our PIFO-approximation technique, a single queue will be used. This showcases one of the most important properties of our algorithm, which is the efficient use of available resources. This can be verified by reading the P4 register queue_level at the simulation end. Only the first queue has been used, and the final level is the time stamp of the latest packet having left the switch.

```
RuntimeCmd: register_read queue_level
queue_level= 159280574, 0, 0, 0, 0, 0, 0, 0
```

It is of crucial importance to check, in first instance, which are the capabilities of the behavioral model in terms of achievable throughput. At the end, we are dealing with software, which is trying to simulate traditionally hardware-based processes and that will enforce some limitations in the performance. Learning those limitations will allow us to control at every moment at which region are we working on: congestion or less than 100% utilization. And more importantly, will allow us to decide the best topology parameters to guarantee, for example, that

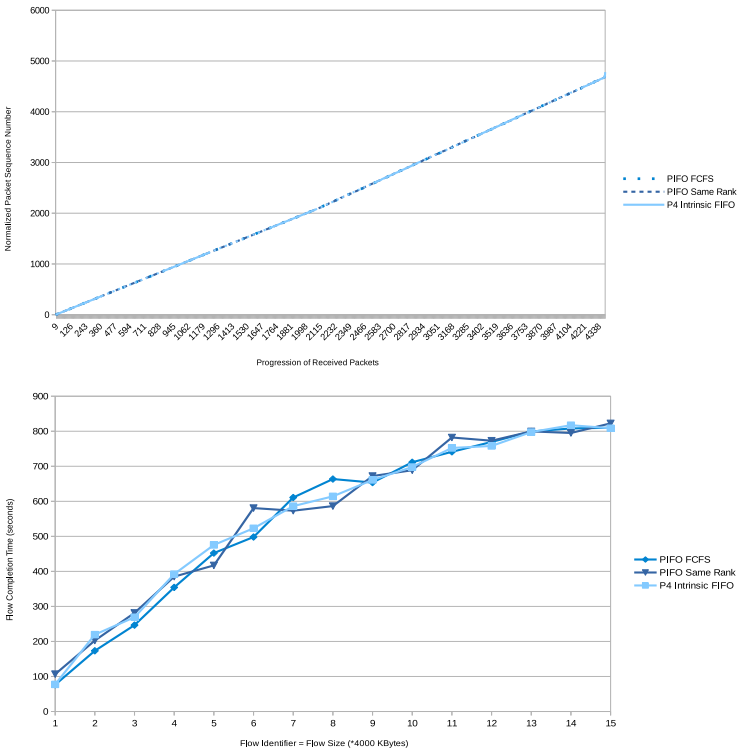


Figure 4.2: FCFS sequence order and flow completion time (FCT) simulation results.

is the output link the one acting as a bottleneck, and not the P4 program CPU limitations. A simple *iperf* test on the initial topology with all the links set to 1000Mbps is enough to obtain the measurement of the P4 software bottleneck. This metric might change with each different P4 program, so it is important to run the experiment for each new scenario simulated. To achieve higher levels of throughput, one recommendation is to disable the logs and pcaps, which can be done when compiling the behavioral-model. In our case, for the FCFS program, we obtain a throughput of 524Mbps. We will set access links to 1000Mbps and output link at 5Mbps to make sure that the latest is the one acting as network bottleneck.

4.3.2 Strict Priority

For priority scheduling, we will follow a similar procedure, analyzing the difference of P4 intrinsic priority scheduling versus our proposal, in terms of granularity, bandwidth sharing and flow completion time. For bandwidth share, the same three-host single-switch topology will be used as reference. For P4-intrinsic priority, 8 different priority queues are configured in the P4 program. Therefore, 8 different traffic flows will be generated from host 1 to host 3. Each flow will have assigned a different traffic priority, which will be specified by taking advantage of the IP ToS field. They will start a UDP connection with the server running on top of host 3, transmitting continuously at a 10Mbps bandwidth. As the output link is bottlenecked to 5Mbps, the switch will experiment congestion, being forced to apply the priority algorithm with which has been configured. To observe the effects of priority, flows will be started in a progressive manner, from lowest to highest priority. The highest priority flow will only transmit during 10 seconds, and then flows will be stopped also progressively, every 10 seconds each, until completing all the connections. The reception bandwidth will be measured at h3, with the objective of analyzing which amount of bandwidth share has been allocated to each flow. The

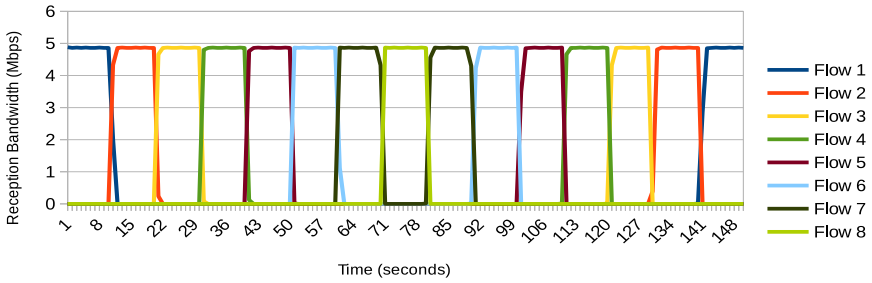


Figure 4.3: Strict priority scheme bandwidth allocation under progressive flow generation.

result can be depicted in Figure 4.3, with the maximum-priority flow at each moment taking a complete portion of the 5Mbps share, not letting lower-priority flows to transmit their required data.

When extending the SP algorithm to the PIFO-abstraction, there is no limitation by the available number of queues anymore. Although having only 8 available queues to use, a higher number of ranks can be specified. The flows corresponding to those ranks, however, will have (if the number of ranks is higher than the number of queues) to share the resources, not enjoying the whole portion of bandwidth share as before. To analyze this situation, we will now simulate the same scenario but with 15 flows instead of 8, and with the PIFO approximation instead of the P4-intrinsic SP. The result can be seen in Figure 4.4. As expected, for a number of flows smaller than 8, the behavior is exactly the same as the P4 intrinsic SP. From the 9th flow on, new connections are forced to share the limited resources.

The 9th flow represents the point in which there is not a unique queue per flow anymore. 9 flows need to share 8 queues. The last arrived flow will always be allocated to the maximum priority queue. However, the previous flow can also reach the top queue when the 7 following flows have set the queue levels in exact increasing order. Thanks to these states, the second highest priority flow is able to achieve a 1Mbps of bandwidth, letting the maximum priority flow only achieve 4Mbps out of the available 5Mbps. Our algorithm can be understood as a fight from low-priority flows to achieve the top queues, against the maximum priority flow (smaller rank), changing queue_levels of the queues below every time one of its packets encounters a lower-priority flow in the highest-priority queue. The following added flows until the 15th one will behave in the same manner with the existing ones. As can be seen, as we add more and more flows, we reach the point in which it gets easier and easier for not highest-priority flows to reach the top queue. In this way, the bandwidth share is more and more equally distributted among flows leading the priority ranking. Flow 15, for example, only achieves 2Mbps, as it has to share the 5Mbps bandwidth with flow 14 (also getting a share of 2Mbps), and Flow 13, which due to the ordering of low priority queue_levels, manages to reach the top queue achieving an average of 1Mbps. It is a great result to observe that PIFO allows an extension of granularity with respect to P4-intrinsic SP.

Interesting debate arises on resetting queue levels to 0 when queues finish draining their packets. As the algorithm is adaptive by itself, there is no need to do it. But not only there is no need, it is worse doing it. Setting each queue to zero once it has finished draining packets allows holes to be created in middle queues, making incoming packets miss-believe that they have reached the top queue, resulting in an inefficient use of the resources. Secondly, setting the queue levels to zero only when all queues have drained their packets goes against the bursty

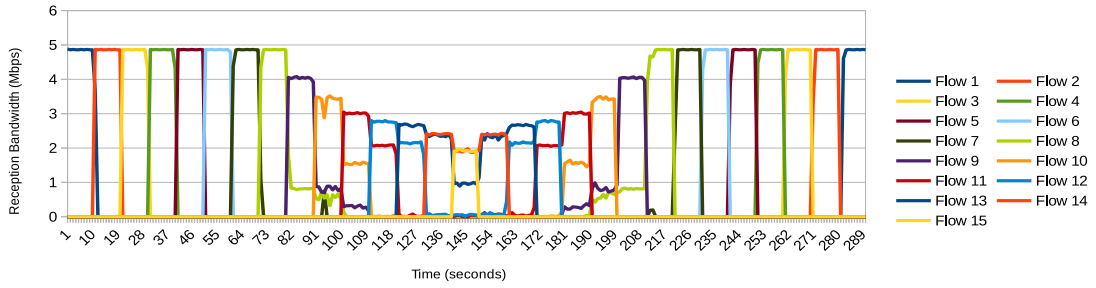


Figure 4.4: SP-PIFO scheme bandwidth allocation under progressive flow generation.

nature of Internet traffic (some time with traffic, queues empty, some time with traffic again, etc). And, at the same time, it forces the system to return to the initial transient state, which is the Achilles heel of our algorithm, due to the amount of blocking it occurs until ranking convergence is reached.

4.3.2.1 Limitations

As one can see from the examples above, the priorization in our PIFO approximation is not completely strict due to a limitation in the resources. When the number of flows is smaller or equal than the number of queues, we can guarantee a unique queue per flow. In this case, and assuming that there is no limitation in the queue rate, the maximum priority flow can gather all the entire available bandwidth share. However, when the number of flows is bigger than the number of queues, PIFO adaptive mechanism forces flows to share the queues. Of course, giving priority to lower rank flows. But this priority is not completely strict, in the sense that lower priority flows, due to the lack of queues, can also make it to the top one (when low priority flows fill the bottom queues). So the highest priority flow has to share queue, and therefore bandwidth, with the lower priority flows that manage to reach the top. This effect has been depicted in Figure 4.5 for better understanding, and is the one suggesting the two questions we will focus on the following lines.

The first one is how can we achieve strict priority in PIFO, when we have more flows than available queues. The solution to that challenge would be leaving the highest priority queue directly for the highest priority flow, and distributing the rest of the queues as PIFO among the lowest priority flows. This way, whenever we want to give maximum priority to a flow, we just need to allocate it to the 'urgent' queue. With this design, the highest priority flow would have a unique queue for itself, being able to enjoy strict priority in the highest manner. And here two different variants arise. When the highest priority flow has finished its transmission, the lower priority flows can transmit following the PIFO approximation. This is the case we have depicted in Figure 4.6. The same initial topology is used but with a bottleneck of 10Mbps and UDP transmissions of 10Mbps as well to make it easier to distinguish the percentage of bandwidth share allocated to each flow. It is important to remark that investing one queue for strict priority also means loosing it from the PIFO approximation of lower priority flows. The second variant is letting the flow with second-highest priority to occupy the 'urgent' queue when the highest-priority flow has finished its transmission. A balance needs to be found, maximizing the queue utilization, but ensuring at the same time that blocking will not occur when 'urgent' flows need it back to transmit. Guaranteeing reservations while keeping high utilization is hard, specially due to the bursty nature of flow connections. Following lines depict the evolution of queue levels as flow with higher priorities (lower rank) start their transmissions.

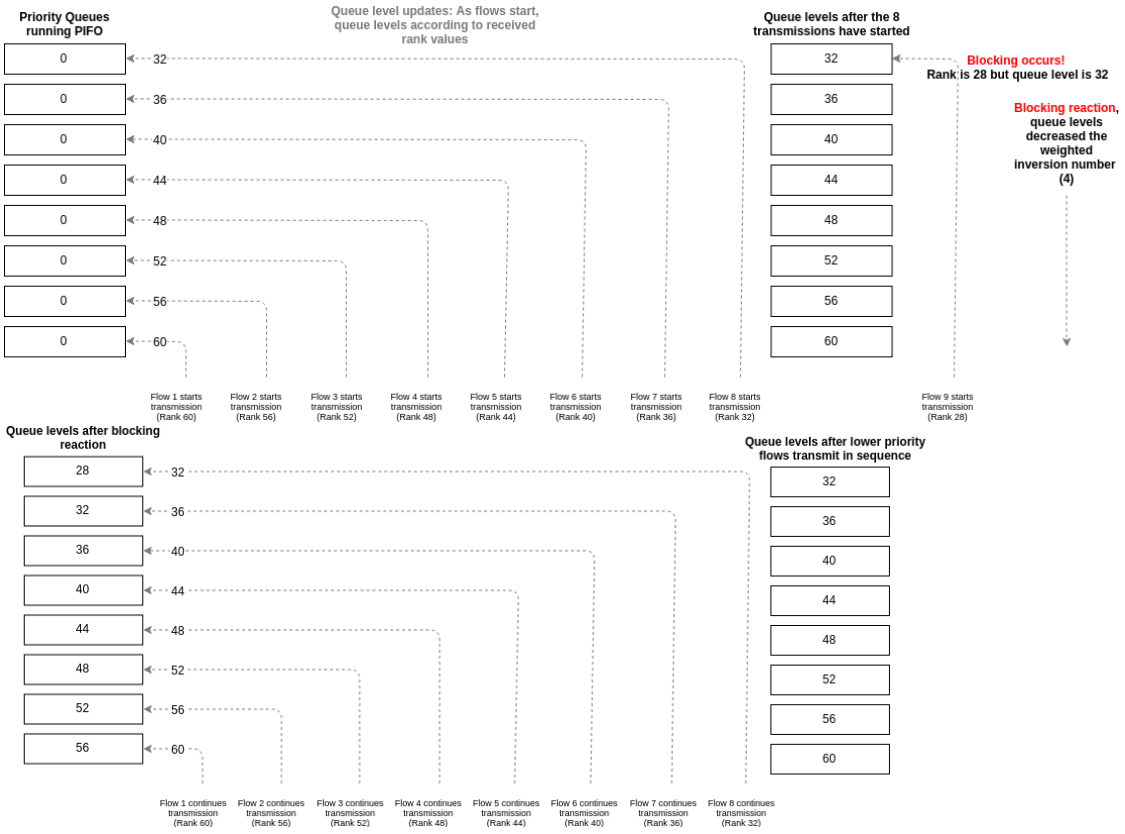


Figure 4.5: Description of blocking reaction effects in SP-PIFO.

```
queue_level= 0,   0,  0,  0,  0,  0,  0,  0
queue_level= 0,   0,  0,  0,  0,  0,  0, 80
queue_level= 80,  0,  0,  0,  0,  0,  0, 76
queue_level= 80, 76,  0,  0,  0,  0,  0, 72
queue_level= 80, 76, 72,  0,  0,  0,  0, 68
queue_level= 80, 76, 72, 68,  0,  0,  0, 64
queue_level= 80, 76, 72, 68,  64, 0,  0, 60
queue_level= 80, 76, 72, 68, 64, 60,  0, 56
queue_level= 80, 76, 72, 68, 64, 60, 56, 52
queue_level= 80, 76, 72, 68, 64, 56, 52, 48
queue_level= 80, 76, 72, 60, 56, 52, 48, 44
queue_level= 80, 76, 72, 68, 60, 56, 48, 40
queue_level= 80, 76, 60, 56, 52, 44, 40, 36
queue_level= 80, 76, 68, 56, 48, 40, 36, 32
queue_level= 80, 76, 68, 56, 48, 40, 32, 28
queue_level= 80, 72, 56, 48, 44, 40, 32, 24
queue_level= 72, 68, 64, 56, 48, 40, 32, 20
queue_level= 68, 60, 44, 40, 36, 28, 20, 16
queue_level= 72, 56, 48, 44, 40, 24, 16, 12
queue_level= 80, 52, 44, 40, 36, 32, 20,  8
queue_level= 80, 76, 72, 48, 40, 32, 16,  4
```

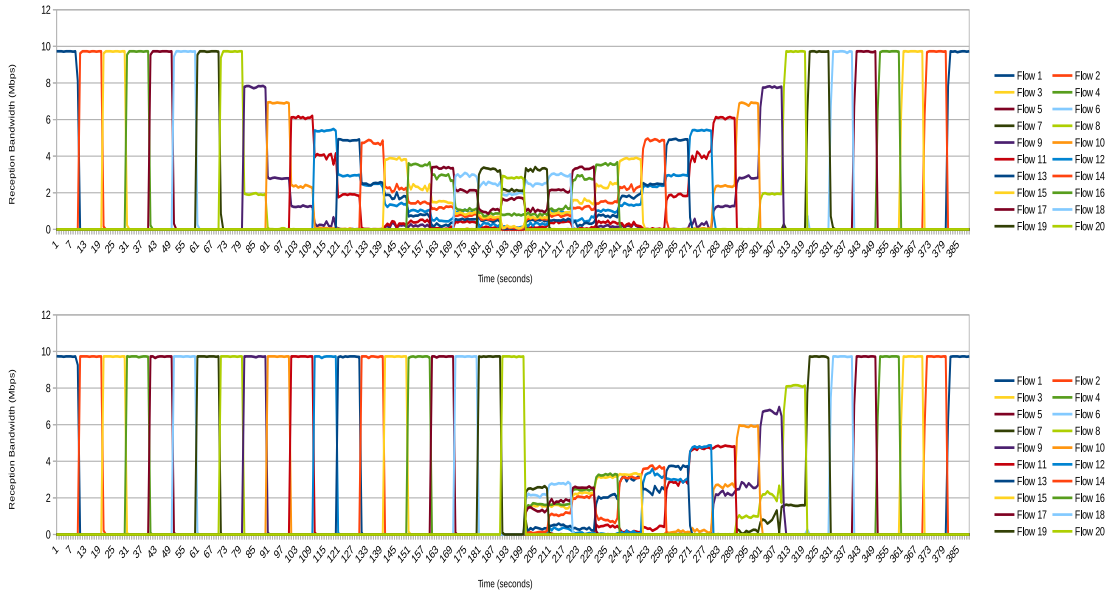


Figure 4.6: SP-PIFO bandwidth allocation on progressive flow generation. Urgent-queue effect.

4.3.2.2 Hijacking the algorithm

In the same way Internet provides flexibility to users around the world to virtually connect and exchange information in a diversified manner, it also offers a broad space for hackers to perform all type of malicious activities one could imagine. For this reason, any new algorithm or protocol designer should spend a non-slight amount of time to analyze possible security flaws and implications. Our case is not an exception. Specially when the logic behind is well known and understood, it is relatively easy to find manners of breaking the algorithm expected behavior or spoofing it to get higher rewards.

Two types of attacks can be differentiated, based on the attacker objective. First, gathering the whole possible link rate for its own profit. Second, blocking the system to prevent flow transmissions from taking place in the output link (i.e. DDoS).

With a solid knowledge of the system state at a specific moment, it is straightforward to attack the algorithm and make it allocate all the available resources to one of your transmissions. Assuming a scenario in which ranks are set from the end-host, the attacker only needs to know which is the minimum rank that non-malicious packets are allowed to specify. With this knowledge, the attacker will be able to place their packets in the highest priority queue by just advertising a rank in their packets with a level below than the regulated ones. Achieving the highest priority queue, as has been seen in the previous sub-section, does not necessarily guarantee high (or the highest in this case) transmission rates, as top queue can be shared by different flows, making them share the available output link capacity. The second step on the attack is then lowering the levels of bottom queues to prevent lower priority packets reach the top queue and enjoying this way, finally, the whole link resources for the attacker benefits. This type of attack is exemplified following the lately presented simulation, considering flow with identification number 20, an attacker. Flow 20 already has been defined with the lowest possible rank in the scenario, but as the number of flows is higher than the number of queues, and the transmissions are all at the same rate, it still has to share the output link with the rest of

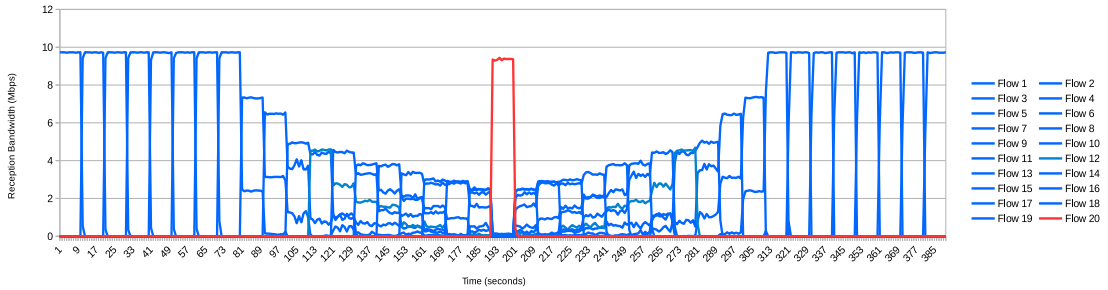


Figure 4.7: SP-PIFO bandwidth allocation on progressive flow generation. Hijacking attack.

the flows. To attack the system, it would just have to increase its transmission rate, for example from the default 10Mbps to 1000Mbps, to ramp up the blocking probability, and decrease this way lower priority queue levels, making higher-rank flows be directed to those bottom queues. With this strategy, the attacker is not only blocking the other flow transmissions, but is enjoying the highest possible bandwidth allocation. The main drawback of this technique, from the attacker perspective, is that the amount of traffic generated to deny the service of other flows is higher than the transmission goodput that the attacker can finally use. In other words, out of the 1000Mbps generated for the attack, only 10Mbps (the bottleneck) will be transmitted successfully to the destination, as the majority will be dropped in the switch. This same strategy can also be used as a denial of service attack, preventing non-malicious flows to travel through a selected link.

When the objective is not to gather the network resources nor to deny transmissions from the switch, but just to break the logic behind traffic differentiation, a different approach can be followed. The rationale in this case is the inverse: trying to drastically increase the levels of low priority queues so that all non-malicious transmissions easily reach the top and collapse together at the highest-priority queue. As a result, one single FIFO queue is left available, breaking all the PIFO logic and making traffic differentiation impossible to be implemented. The attack execution is a bit more complex, as more than one flow transmission is required. In particular, N-1 attacking-flows are required, where N is the number of PIFO queues, each one destined to block one of the low priority queues. All flows will have assigned a rank value higher than the maximum regulated (meaning the rank scale non-malicious traffic use)rank and different from the other attacking-flows. By keeping those flows transmitting at a high rate, the attacker will minimize the effects of blocking reactions, making all non malicious traffic collide at the highest-priority queue. In this type of attack the malicious host is not obtaining any reward aside of breaking the traffic differentiation and priorization that PIFO would be providing. Although these attacks are intrinsic for the PIFO implementation, and therefore can be used for all scheduling alternatives, depending on the application running on top, more or less deeper effects can take place.

4.3.3 Shortest Flow First: Minimizing average flow completion times (FCT)

There have been a number of algorithms proposing alternatives to diminish flow completion times. Shortest Remaining Processing Time (SRPT) has been proved to be optimal for minimizing average FCT when scheduling over a single link, prioritizing flows in order of remaining flow size. As the determining the deadline is not straightforward, several simplifications have been presented. The Shortest Flow First approximation, based on using the absolute flow size instead of the remaining flow size, has been shown to provide close-to-optimal results [AYS+13].

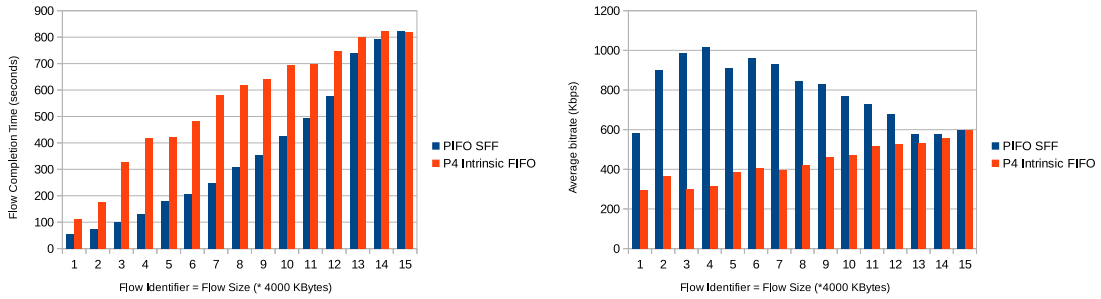


Figure 4.8: Shortest Flow First (SFF) Flow Completion Time (FCT) and average bitrate results.

In this section we will analyze if it is possible to reduce flow completion times by using these strategies under PIFO abstraction. As the pursuit is to measure flow completion times and not strict bandwidth share, we will use TCP connections. For this experiment D-ITG [BDP12] will be used, a traffic generator with more customizable configurations than *iperf*. 15 flows will be created. But this time with two differences respect the previous simulation. Every flow will have different sizes. We define the size of the flow as the amount of data that has to be transmitted. The packet size and transmission rate will be defined by TCP. All of them will start their connections at the same time. And their rank will be set as the size of each flow, making use again of the ToS field to carry the rank information from the end-host to the switch. As in TCP transmissions ToS fields have to be multiple of 4 (the two lowest bits are reserved), the size of the flows will be set accordingly. Note that in the TCP protocol implementation, some packets are forced to be transmitted with ToS equal to 0, which has to be also considered in the PIFO implementation.

```
hdr.ipv4.tos = flow.size;
meta.rank = (bit<32>)hdr.ipv4.tos;
```

Simulation results describe an average flow completion time in SFF is of 367.1148228667 seconds, while in FIFO remains 556.7540384 seconds. It is interesting to see the huge benefits that can be obtained by simply prioritizing small flows over big ones. The separation of big flows, also often called elephants, from small flows, often called mice, has been an objective of many areas inside the networking field. This differentiation, for instance, has been the basis of advanced circuit building and load balancing techniques, which have aimed to generate orthogonal paths for each flow type in the network (as much as possible), to improve global quality of service. The performed simulation can be taken as an illustrative example more demonstrating how important can it be to define differentiated treatment for different characteristics traffic flows.

Is SFF PIFO-approximation the best rank allocation that we could achieve to minimize FCT for this particular scenario? Or, from another view, could we distribute packets from traffic flows in another manner among priority queues to achieve a lower average FCT under the same architecture? The answer to the last question is clearly yes. Being aware of the traffic being transmitted, one could adjust the allocation of traffic directly to SP queues to achieve better performances. For the particular case occupying us in this moment, if we know that we are dealing with 15 traffic flows of different sizes and we want to minimize FCT, we could avoid the transient of our algorithm by grouping consecutive traffic flows in pairs and allocating them directly to the queues in order of priority. The results of a manual mapping of flows to

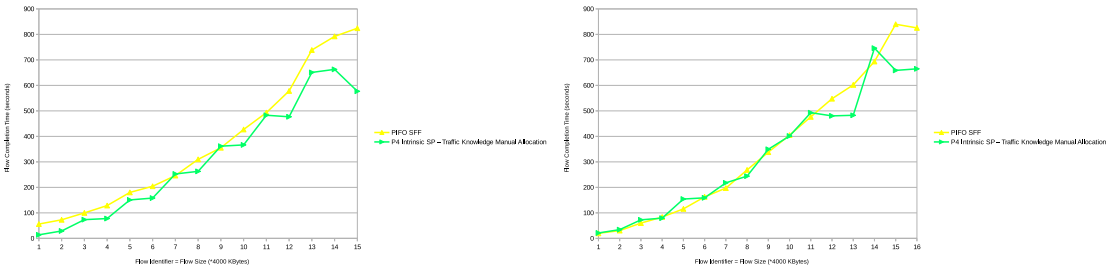


Figure 4.9: Shortest Flow First (SFF) and manual allocation under traffic knowledge FCT results.

strict priority queues can be seen in Figure 4.9. Taking the manual allocation as the ground truth, or the best possible reference strategy, it can be seen how our dynamic algorithm is quite close to the optimal one. The good point of our solution is that previous knowledge of the scenario or the traffic distribution is not required in advance. The algorithm adapts to changes. If the scheduling objective is suddenly switched, after a small transient time our algorithm will adapt. What we achieve in a dynamic manner without previous knowledge of the traffic can be improved when this knowledge is available. But then a different algorithm is required for each particular scenario, and that is not the challenge we are willing to solve. The improvement in the last flow is just due to the fact that it has a queue allocated for itself, so that it can gain access to the entire queue rate.

4.3.3.1 Adaptiveness

One of the great points of our implementation is the ability to quickly re-adapt from a set of ranks to another. This characteristic is very powerful specially in those scenarios where ranks can be set by the end-hosts. Without need to reboot the switches on the path, a simple swap in the ranking definition will, after some transient time, result in a completely different output behavior. Imagine that a switch on a company network is carrying real-time transmission packets, with small ranks ranging from 50 to 59. In a certain moment, the company decides to use the link for a data center storage backup maintenances, with longer flows and bigger ranks, say from 10 to 90. In a normal switch with strict priority, a technician should manually remap the priority of those flows in each switch along the path. With our algorithm is just enough to start transmitting the new types of traffic with their updated ranks. Figure 4.10 depicts this adaptiveness of the PIFO approximation, in a scenario that starts from a switch reboot, with all the queue levels set to zero, to a transmission of several packets of flows ranked from 50 to 59, and finally to a rank update from the end host, which stops the previous transmission and starts sending packets of flows ranked from 10 to 90. It can be clearly seen how the algorithm adapts in a rapid manner, first changing the queue levels to the ranks of the first packets arriving after performing the swap (ranks 10, 20 and 30), and progressively, as higher-ranked packets start to come, lower priority queues adapt as well to accommodate those new entries. It is worth mentioning, that the change occurs with such rapidness, that we had to perform simulations limiting the access link at 1Mbps and the output at 0.5Mbps to be capable of capturing the evolution of different queue levels with sufficient detail.

4.3.3.2 Scalability

Being this simulation the first in which a non mono-atomic increase arrival of ranks is handled, it is the right moment to pose an important question: Which is the limitation in terms of

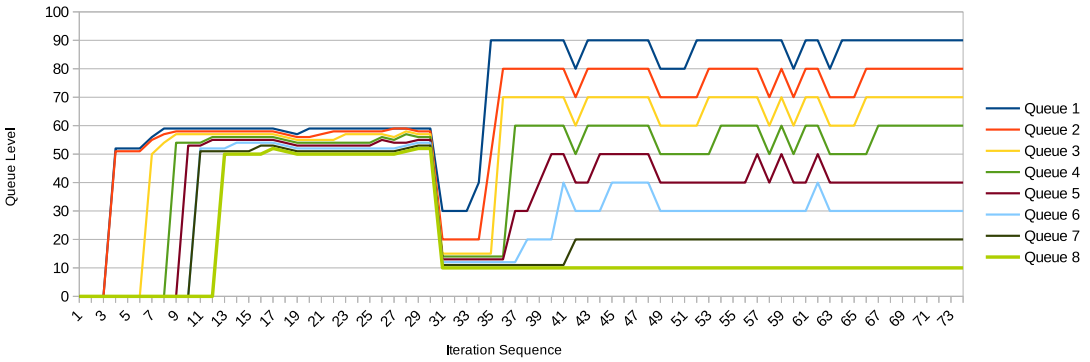


Figure 4.10: SP-PIFO adaptiveness performance.

rank variability that the system is able to manage? It is clear that with 8 queues and 15 flows, the results obtained are quite promising. But what happens when the number of flows is not fifteen but a couple of hundreds or even thousands. Will the algorithm still provide any beneficial results? In fact, the number of flows is not what matters exactly, as having a very big number of flows sharing a small range of defined ranks is completely equivalent (from the PIFO implementation perspective) to having a single-but-very-long individual flow per each rank. The relevant parameter to take into account when considering scalability is the number of different ranks. The higher the number of ranks, the more difficult it will get for the scheduler to map them to the available strict priority queues. As has been shown in Section 3 of the thesis, the higher the number of ranks, the less-accurate the PIFO-approximation can be.

Figure 4.11 depicts the result of transmitting 254 differently-ranked flows from h1 to h3. In this case we used the TTL field to mark the packets from the end-host. This is the maximum number of flows that D-ITG allows to transmit in one session. The bottleneck in this case has been switched to 80Mbps, in order to let the first packets of all flows reach the destination on time. If not, D-ITG drops the connection as soon as the source does not receive the first acknowledgement. We can see how the adaptiveness of the PIFO approximation allows the algorithm to improve the default FIFO performance with a high level of robustness. Resulting average flow completion times are 133.296 seconds for SFF versus 288.871 seconds in P4-FIFO. The intuition behind is that small flows will always go to the higher queues, no matter how many bigger flows are competing as well. As soon as short flows finish transmitting, bigger flows will move to higher priority queues, and progressively all flows will go from bottom queues to the top ones in order of rank. Finally, as more flows finish sending data, even the bigger ones will be able to reach the top, completing their transmissions as well. Again, this shows that scalability is not an issue for our implementation. As mentioned before, the main Achilles heel of the proposed algorithm is the starvation of higher rank flows while a huge amount of small flows want to transmit as well. In this same scenario, if small flows were transmitting in a continuous manner (assume infinite flow size), bigger flows would never reach the top, and their services would be completely denied.

4.3.3.3 Traffic distributions

The last experiment of SFF consists on analyzing how the algorithm reacts to different type of traffic loads. With the fix number of 15 flows, we will change their characteristics in the following manner. Let's define small flows as those ranging from 400 to 6000KBytes. And big

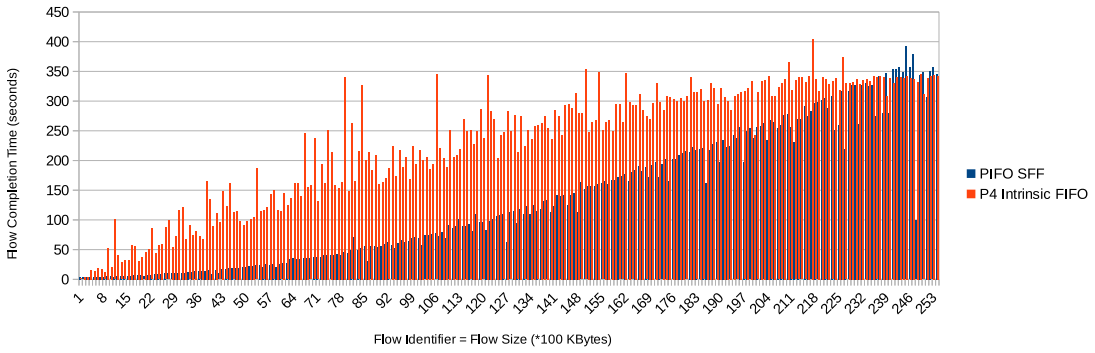


Figure 4.11: SP-PIFO scalability performance.

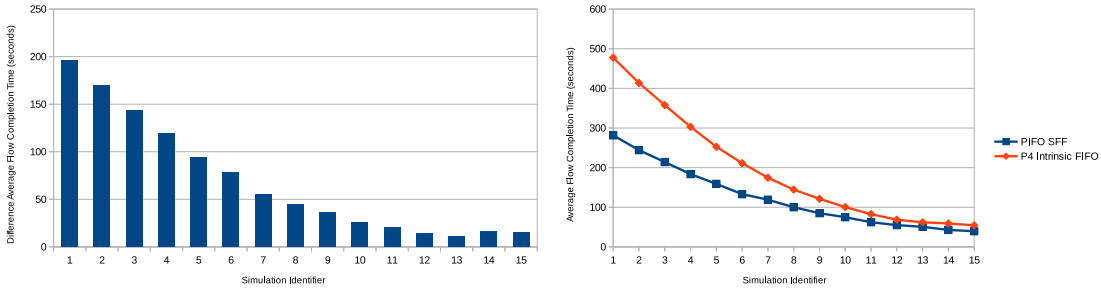


Figure 4.12: SFF vs. P4-FIFO Average Flow Completion Time difference under progressively increasing percentage of small flows.

flows from 20000KBytes to 25200KBytes, following the limitations of D-ITG and ToS field for rank specification. The first simulation will consist on a small flow and 14 big ones, with final flow sizes of {400, 20000, 20400, 20800, 21200, 21600, ..., 25200}. In the second simulation, we will remove the biggest one, to add the following smallest one: {400, 800, 20000, 20400, 20800, 21200, 21600, ..., 24800} and we will proceed in the same manner until the final simulation, which will have flows sized as {400, 800, 1200, 1600, ..., 6000}. We can see how SFF outperforms P4 Intrinsic FIFO for all workloads, making a special difference as traffic load gets bigger. This makes our result to be in complete consonance with [OS15].

4.3.4 Weighted Fair Queuing: Achieving short and long term fairness

Fairness is probably one of the most-wanted objectives when performing scheduling. Usually is considered under the max-min fairness criteria, defining a fair allocation as one in which no flow can get better treatment without hurting the performance of another flow with a lower level of service. In a more formal definition, it can be stated that "an allocation is fair if (1) no user receives more than its request, (2) no other allocation scheme satisfying condition 1 has a higher minimum allocation, and (3) condition 2 remains recursively true as we remove the minimal user and reduce the total resource accordingly" [Bal16].

As has been reviewed in previous sections, obtaining an allocation that satisfies max-min fairness among packets of different flows is not an easy task. Various techniques have been proposed over the years, each with specific implementation details.

- **Round Robin** [HG86]: This approach provides max-min fairness only for scenarios with fixed packet lengths. Ideally, it requires an individual queue per flow, from which the scheduler can drain packets at a one-packet-per-queue basis, iterating over the queues in a sequential manner. With this idea, a packet from each backlogged queue is drained at each round, allocating fair bandwidth shares to each contending flow. As having a physical queue per each flow is not realistic in practice, it is a common idea to map traffic flows into different classes. With each class having a dedicated queue, Round Robin scheduler just scans class queues in order, serving one packet from each non-empty queue at each round. With this solution, max-min fairness and protection is achieved among classes. More advanced proposals suggest replacing physical queues by state-saving on the switch. Virtual queues are generated by keeping track of some metrics like the virtual time and the instantaneous round number. For a very basic Round Robin implementation on P4, based on our PIFO approximation and the virtual queues methodology, it is enough to define counters tracking the number of packets sent by each flow. Flows with lower number of packets transmitted will have lower ranks, and therefore priority over flows with very steady connections. This first approach works for fixed packet length scenarios in which all flow transmissions start at the same time. However, real traffic usually operates by bursts, with some empty intervals between consecutive transmissions. In this case, the most common technique is to keep track of the round number. When new connections take place, instead of starting their packet counter from zero, they start from the current round number, preventing them from having complete priority over existing connections. Round Robin is still the root of a great majority of scheduling algorithms aiming to provide fairness. Most of their successors still keep the idea of per-flow virtual queues, trying to emulate the original Round Robin procedure, and keep track of the virtual round value to accommodate the bursty nature of traffic streams.

- **Weighted Round Robin and Variable-length Round Robin** [Sem01]: Traditional Round Robin is unfair if different flows or classes are given distinct weights. Assume the case in which 5 different classes are sharing the output link resources in a Round Robin manner, but we want the scheduler to allocate twice the fair proportion of resources to the first class, keeping the rest with the original distribution. Weighted Round Robin considers this scenario by modifying the basic Round Robin at the time of updating packet counters. Updating all class counters in steps of 2, except the first class counter, updated 1 by 1, the number of resources allocated to first class are directly doubled from the rest. Traditional Round Robin is also unfair if packets have different lengths. In this case, draining one packet per queue can mean allocating different number of resources to each class or flow. The appropriate modification for those scenarios in this case is increasing the counter not by one unit each time a packet is sent, but by the actual length of the packet. All those modifications are just updates from the original Round Robin scheduling algorithm, and they provide backwards compatibility (i.e. Variable-Length Weighted Round Robin will just work as a simple Round Robin if only fixed-length packets with same weight are considered).

- **Generalized Processor Sharing** [PG93]: This new branch of research proposed studying the opportunities of Round Robin from another perspective. They stated that ideal fairness could be found if an infinitesimal amount of data could be served from each flow queue at each round. As that is not implementable, they proposed different approximations to that model, usually referenced as Fluid Flow Fair Queuing or just Fluid Round Robin. When the infinitesimal amount is translated to 1 bit, then we have the bit-by-bit

Round Robin. Although none of those algorithms are realizable in practice, they serve as a theoretical basis for the upcoming (those yes implementable) approximations.

- **Weighted Fair Queuing** [DKS89]: The biggest and most famous GPS approximation, also known as packetized GPS or packet-by-packet GPS, suggests computing the finishing transmission time of packets as if they were transmitted through the bit-by-bit fluid model, and scheduling them in order of increasing finishing times. To do it, the notion of round number and finishing time are used. The round number (also called virtual time) is the number of virtual rounds completed under the fluid model, assuming that each round a bit is served from an active connection. Finish time is the round number at which a packet would finish transmission if it was transmitted with the bit-by-bit model. The idea is keeping track of those virtual variables (round and finishing time), and scheduling packets in order of finishing times. For the simple case in which flow virtual queues are continuously backlogged, the finish time of a packet k from flow i is equal to the finish time of the previous packet from the same flow, plus the transmission time of the current packet under the bit-by-bit basis.

$$F_i^k = F_i^{k-1} + \frac{L_i^k}{\phi_i} \tag{4.1}$$

For a more complex case considering that flows are not constantly transmitting, but can be idle for some time, finishing time can be accurately computed with the help of the round number. In case the queue is backlogged, the finishing time will be selected as in 4.1. Otherwise, the round number will be taken as the start transmission time:

$$F_i^k = max(F_i^{k-1}, R(t)) + \frac{L_i^k}{\phi_i} \tag{4.2}$$

Although, on paper, weighted fair queuing is the most accurate algorithm for achieving max-min fairness, a precise computation of the round numbers is highly complex. Several variants have appeared making implementation details a bit easier.

- **Self Clocked Fair Queuing and Start Time Fair Queuing** [Sah08] [Gol94]: In SCFQ, the round number is approximated to the finishing time of the last packet being served. By just updating the round number every time a new packet is scheduled, finishing time can be easily computed. The price to pay is an increase worst case delay.

$$F_i^k = max(F_i^{k-1}, CF) + \frac{L_i^k}{\phi_i} \tag{4.3}$$

Start time fair queuing aims to be more precise by working with both, start time and finishing time. The start time of a packet arriving into an inactive connection is the round number at the time the packet enters the switch. If it arrives into an active connection, meaning some previous packets are already in the queue waiting to be transmitted, then the start time is the finishing time of the previous packet. Packets are scheduled in order of start time and the round number is updated as the start time of the last packet being served. Other options like WF^2Q aim to reduce the WFQ burstiness derived from the fact that flows need to wait for other flows to transmit before having consecutive access to the channel. And they do it by selecting at each scheduling decision packets only from the ones which would have started transmitting under the fluid model, instead of having to select among all available packets.

To analyze how fairness can be implemented on our PIFO abstraction, we will focus on STFQ algorithm, for being the most accurate WFQ approximation. As can be seen, it is the first case in which the ranking definition needs to be configured from the switch itself, as local information is required, like the virtual time, and contributions from different flows need to be taken into account. Bandwidth share distributions will be evaluated, when various UDP transmissions are started demanding different rates. Obtained results for the main scenarios are depicted in Figure 4.13. In the first example, the sum of bandwidth demands is below the capacity and therefore all requirements are met. In the second example the capacity is exceeded with requirements of $\{0.5, 2, 4\}$Mbps. How should a max-min fair algorithm allocate bandwidth to the links? According to the max-min criteria: $5/3 = 1.66$ would represent a bandwidth distribution of $\{1.66, 1.66, 1.66\}$Mbps to each flow. As 1.66 overfeeds the demands of flow 1, which only needs 0.5, the excess $1.66 - 0.5 = 1.16$ is divided among the other remaining flows, $1.16/2 = 0.58$, obtaining a distribution of $\{0.5, 2.24, 2.24\}$Mbps. But the second flow only needs 2. So we have an excess of 0.24, which will go to the third flow, being the final max-min fair distribution: $\{0.5, 2, 2.48\}$Mbps respectively, which is exactly what we obtain with our algorithm. But now let's move to the important scenario. Example 3 shows how the algorithm responds under the requests of $\{0.5, 2.5, 4\}$Mbps. In this case we don't achieve the max-min fair distribution of $\{0.5, 2.24, 2.24\}$Mbps. Instead, what we obtain is $\{0.5, 2, 2.5\}$Mbps. STFQ protects small rates from big ones in a strict sense, giving them complete priority. Although for most cases this provides max-min shares, forcing users with high rates to decrease their transmissions, one can find situations in which a simple difference of 0.1Mbps can suppose starvation of a slightly highly demanding flow in front of the lower one. For instance, in a case where the requirements are $\{2.5, 2.5, 2.51\}$Mbps, the two first flows would achieve their objectives while the third one would not even transmit. Although prioritizing lower rate flows it is already a big milestone, automatically penalizing malicious input streams, not allow misbehaving sources to get a higher share of the bandwidth or to generate delay on other flows by just transmitting packets at a higher rate, in some particular cases that can derive into unfair metrics.

The explanation for this behavior arises from two main problems in the original STFQ adaptation to the PIFO abstraction. The first and most important one is that ranks, as defined originally, depend on packets arriving the switch, not served, under the consideration that all packets scheduled will be transmitted at some point. From our approximation point of view, a packet entering to the switch does not necessarily mean that this packet will be transmitted after a short time interval. Packets with low assigned levels of priority can be starved in lower priority queues during long time in congestion periods. And they can even be dropped after having been processed by the ingress. Therefore, enforcing fairness by considering how packets enter the switch (and not leave), is not the most adequate approach when working on strict priority queues. Instead, the number of packets really transmitted into the output link should be the ones controlling how fair shares are accomplished in our algorithm. At the same time, a lot of new packets can arrive and cross the ingress during the time in which a packet moves from ingress to egress. Therefore, the state updates should be only performed in one of the two, to avoid different flows having access to different values of the same variable. An algorithm scheduling new packets in reference to the amount of bytes really transmitted, would be closer to monitor and react on how fair the algorithm is behaving. Again, there is still a gap between the ingress and the egress. Scheduling according only to the packets being transmitted, means loosing the view of all those packets already in the queue. This will provide some variability on the short term, but on the long term, the desired fair bandwidth shares should be finally obtained.

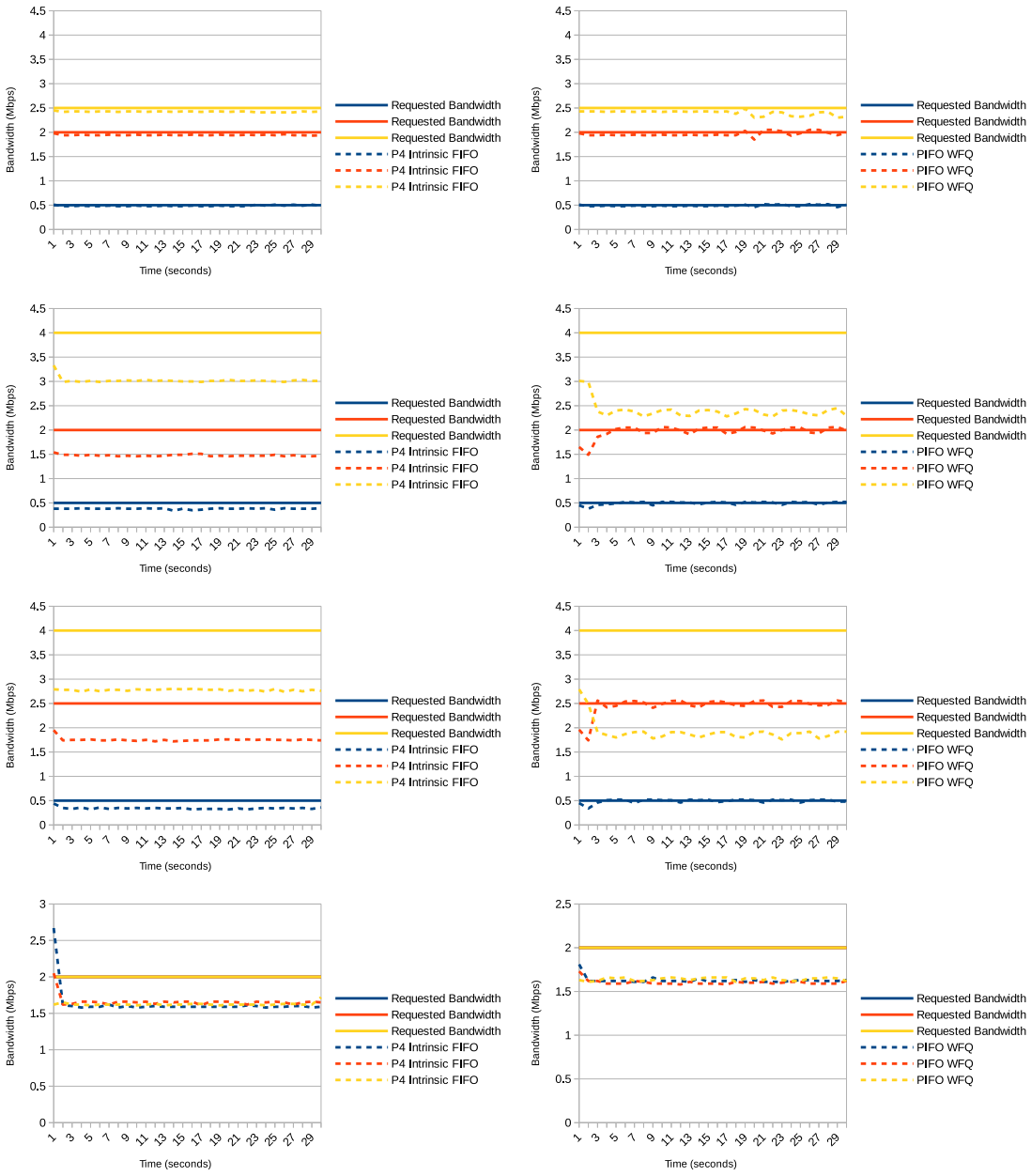


Figure 4.13: Bandwidth share in Original Start Time Fair Queuing 5Mbps Bottleneck.

4.3.4.1 Limitations and alternatives

Following this idea, and aiming to obtaining the purest definition of max-min fairness, we adapted the original version of Start Time Fair Queuing to schedule the packets according to the output rate achieved by each flow. In our proposal, we schedule packets in increasing order of start time. But now the start time of a packet (and its rank) is defined as the last finishing time of the previous packet served (instead of scheduled) from the same flow. In case the flow queue is not backlogged, the round number is used, defined as the start time of the last packet which started transmission among all the flows. It is of crucial importance being aware of the difference between start times and finishing times. Not doing so, can result in scenarios in which the round number is kept higher than the finishing times continuously, generating a synchronism in which big flows would gather all the resources available in a continuous manner. While the pseudo-code for the original STFQ was already introduced in Section 3.2, the proposed algorithm modification is described in the lines below.

Algorithm 5 Ingress processing

```
1: finishing_time.read(meta.last_finishing_time, hdr.ipv4.tos);
2: round.read(meta.round_time, 0);
3: if (meta.last_finishing_time > meta.round_time) then
4:     meta.rank = meta.last_finishing_time;
5: else
6:     meta.rank = meta.round_time;
7: end if
8: pifo.apply(meta.rank);
```

Algorithm 6 Egress processing

```
1: finishing_time.read(meta.last_finishing_time, hdr.ipv4.tos);
2: if (meta.last_finishing_time > meta.round_time) then
3:     finishing_time.write(hdr.ipv4.tos, meta.last_finishing_time + hdr.ipv4.totalLen);
4:     round.write(0, meta.last_finishing_time);
5: else
6:     finishing_time.write(hdr.ipv4.tos, meta.round_time + hdr.ipv4.totalLen);
7:     round.write(0, meta.round_time);
8: end if
```

When performing the same experiments with the proposed alternative, results in Figure 4.14 are obtained. Similar metrics to STFQ are achieved in general, except in the critical cases, where closer results to max-min fairness are observed. The main drawback that can be easily perceived is the variability produced by the ingress and egress decoupling. The way UDP traffic is generated by *iperf* also accentuates this variation. Further than that, those algorithms are just approximations, the round time is a mere estimation, and therefore gives room to deviations from the theoretical distribution. A good balance of this variation has to be found with the queue depth selection. A very deep queue will decouple too much the ingress from the egress, as there will be a huge gap between the timings of packets entering and leaving. A very small queue will not provide the scheduler with enough space to position the packets accordingly. Our simulations have been carried with the 64 packets size per queue, value default of latest P4 version. An order of 100 is appropriate for the testing scenarios we are working with. When extending Example 3 over 10 minutes, we can observe that our modified version of STFQ gets closer to the max-min criteria than the original algorithm. Results are shown in Table 4.1.

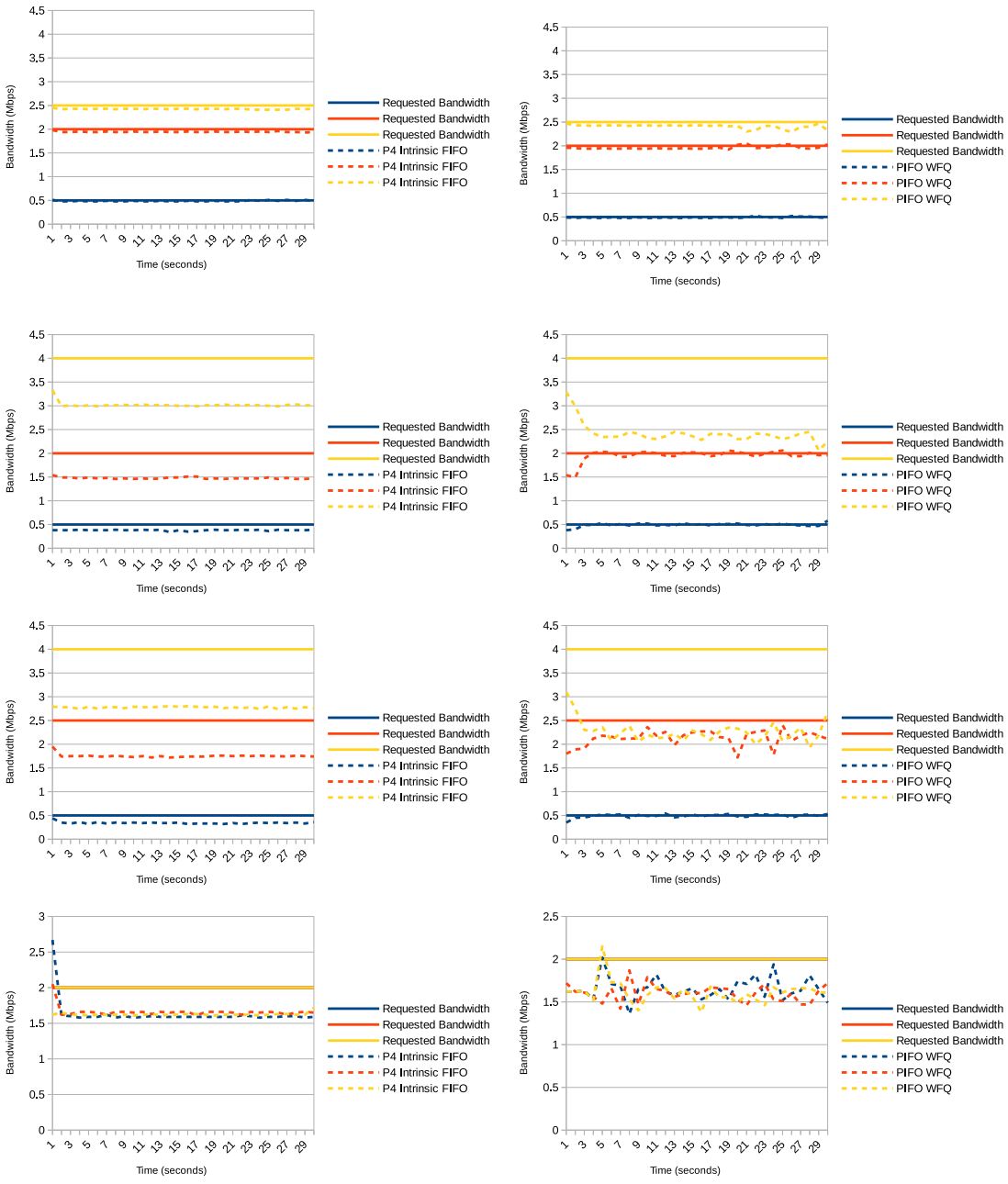


Figure 4.14: Bandwidth share in Modified Start Time Fair Queuing 5Mbps Bottleneck.

Bandwidth Distribution			
Requirements	Max-min fairness	STFQ	Modified STFQ
0.5 Mbps	0.5	0.5	0.497
2.5 Mbps	2.24	2.5	2.15
4 Mbps	2.24	1.87	2.21

Table 4.1: Average throughput (Mbps) achieved on 10 minutes third example simulation.

4.3.4.2 Reacting to new connections

One of the main characteristics of the presented WFQ approximations is the incorporation of a virtual time counter or round number. This counter allows new connections to define a correct start and finish time for their new packets, when it is the first time they take place, or when long time has elapsed since the last transmission. Figure 4.15 describes this demeanor, when Example 3 connections, sharing the 5Mbps as 0.5, 2.24 and 2.24Mbps respectively, see their transmissions affected when a new demand of 4Mbps arrives, seeing how the resources are rearranged to 0.5, 1.6, 1.6 and 1.6Mbps approximately to keep the fairness in the distribution.

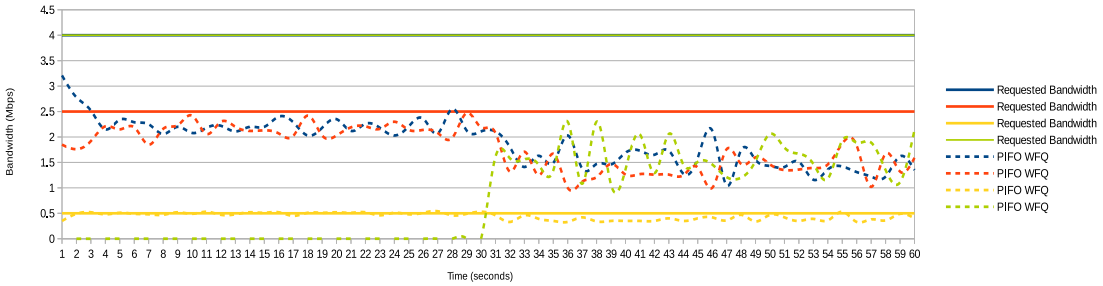


Figure 4.15: Bandwidth share under reaction of new flows.

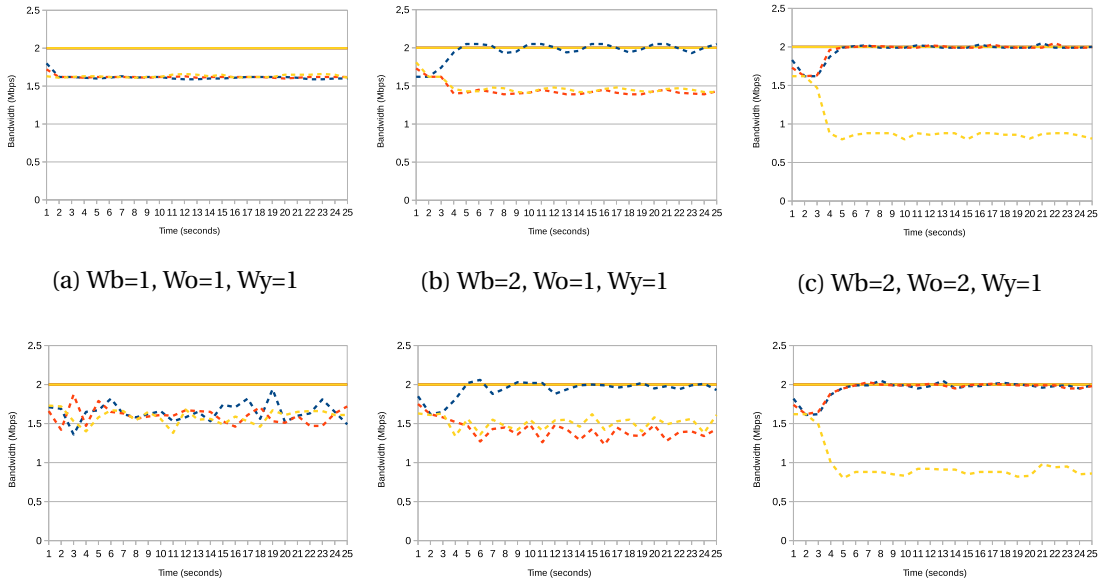
4.3.4.3 Assigning weights

Another interesting feature of WFQ algorithm is the possibility of defining in advance, the weight or portion of the bandwidth that different flows should obtain. This weight definition is performed when selecting the finishing time, in which the length of the packet is divided by the weight accordingly.

$$Finishing\_time = Start\_time + Packet\_size/Rate_i \tag{4.4}$$

$$Rate_i := \frac{w_i}{(w_1 + w_2 + ... + w_N)} * Capacity. \tag{4.5}$$

Departing from the last example in the non-weighted simulation, in which three flows demands of 2Mbps shared the 5Mbps link at a rate of 1.66Mbps each, Figure 4.16 describes the effects of giving more priority to one (and two) of the flows, respectively, by assigning different weights. It is important to remark, that due to the variability produced in our STFQ modification, stronger weights were required to achieve the equivalent results. Instead of using the weight set {2,2,1}, weights {4,4,1} had to be chosen in order to achieve the desired throughput distribution.



(a) Wb=1, Wo=1, Wy=1 (b) Wb=2, Wo=1, Wy=1 (c) Wb=2, Wo=2, Wy=1

(d) Wb=1, Wo=1, Wy=1 (e) Wb=2, Wo=1, Wy=1 (f) Wb=4, Wo=4, Wy=1

Figure 4.16: Bandwidth share modification by setting weights in Original Start Time Fair Queuing (first row), and Modified Start Time Fair Queuing (second row).

4.3.4.4 Limitations

The main concern of WFQ is the scalability limitation (if the number of flows increases considerably). In the PIFO approximation, that would suppose an increase of the number of registers to use. That is why native WFQ is not available for high-speed interfaces. The traditional solution is class-based weighted fair queuing or clustering flows in traffic classes. The same can be directly applied to PIFO implementation. By using hashes, different flows can be mapped to the same finishing_time registers, or traffic classes can directly be described by the end-host through the ToS field. Hash functions are the first consideration when needed to scale the algorithm. Obviously, the granularity provided with SP-PIFO will be limited by the number of available queues. But that is a limitation of the PIFO approximation itself, which has already been discussed in previous sections. The second limitation to consider is given by the length of the register storing the round number. As more and more packets are transmitted, the round number increase rate can rise drastically, making it reach its physical limitations given by the size of its register. One should consider resetting the registers value (to zero) when the round number reaches its maximum limit, setting also simultaneously the stored values of flows finishing_time to zero. This can result in a performance limitation, which is intrinsic to WFQ nature. The amount of memory needed to implement WFQ has been a broadly discussed and investigated limitation since the first proposal arose.

4.4 Multi-hop algorithms

4.4.1 FIFO+: Reducing tail packet delays

The idea behind FIFO+ [CSZ92] is to give priority to those packets having experienced higher levels of queuing delay in previous hops, with the objective of reducing tail latencies, although

increasing a little bit the global average packet delay. FIFO+ can be implemented by marking an equal deadline in all packets from the end-host or the access-switch, and decreasing this deadline as packets traverse switches along the path, with the amount of time that they have spent on the queues. Packets are scheduled according to this deadline. It is important to remark that the scheduling is performed at the ingress, while the time spent in queue can only be accessed from the egress. For this reason, FIFO+ makes no effect in a single switch topology, as all the packets enter the ingress with the same deadline, and no differentiation among them is then possible when performing the scheduling.

FIFO+ makes sense in a multi-hop architecture. But not all architectures are valid. In a linear topology like the one depicted in Figure 4.19, but with all backbone links set to 5Mbps while leaving access links at 1000Mbps, supposing that all switches support FIFO+, and that only h1 is transmitting to h6, two scenarios can occur. If h1 transmits at a rate below 5Mbps, the packets will travel along the path from h1 to h6 without creating congestion in any link, leaving all queues almost empty and experiencing a queuing delay practically null. When there is no congestion happening, scheduling algorithms can not take place. If, otherwise, h1 transmits at 5.5Mbps, congestion will take place at s1, and therefore scheduling will enter into action. However, as s1 is the first hop that h1-packets encounter in the path, all the packet deadlines will be still at initialization value. PIFO application will be useless, performing as a simple FIFO, as all the packets will still be carrying the same rank. It will be at the egress, where s1 will update packet deadlines with the different times spent in the queue and then, they will be ready to receive distinct treatment by FIFO+ in the following hops. However, in this particular case, as h1 is the only host transmitting in the scenario, and switch s1 has already filtered h1 traffic, shaping the output rate at 5Mbps, if following links have also capacities of 5Mbps, they will not act as bottlenecks and packets will not experience congestion in them. Queues will again remain empty and schedulers supported by the switches will not make effect.

This toy example allows us to depict the need of generating consecutive congestion in links, if the FIFO+ differentiated treatment is willing to be observed. There are two possibilities to see the effects of this algorithm. One could be generating consecutive bottlenecks, for instance s1-s2 link at 5Mbps, s2-s3 at 4Mbps, and progressively decreasing the capacity of following links to keep congestion at each switch, and the other is adding new traffic at each hop. Throughout this subsection, we will use both techniques, to analyze and understand how FIFO+ affects flow transmissions, both for transmissions on a single-path and for cases where different route connections encounter.

4.4.1.1 Understanding delay

In order to be able to deeply understand the results of FIFO+ simulations, is important to devote some time to underline which are the main contributions of queuing delay that can play a significant role in the architecture we are working on. As mentioned previously in this thesis, we are using simple-switch, a virtual implementation of the PISA architecture. Our simple-switch instance consists on 64 input ports, each of those with an input buffer of 1024 packets capacity, where traffic coming from the input interfaces can adapt to the ingress pipeline processing rate. Packets then traverse the parser, which recognizes the main headers and metadata, and then they jump into the ingress, consisting on up to 12 stages, each formed by a sequence of match action tables which define the exact manner in which packets should be processed. Is in the ingress where all forwarding is specified, selecting the correct ports to transmit the packets to, and the right queues to use. All this information is carried through special metadata to the

traffic manager, which will enqueue the packets following the policies defined at the ingress. The traffic manager is therefore the one containing our PIFO structure, with the 8 different priority queues, and scheduling packets according to the information received from the ingress pipeline. Each priority queue has a capacity of 64 packets, and once they are completely full, tail-drop is activated. After packets depart the traffic manager, they cross the egress and the deparser, and they are finally forwarded. However the traffic manager is not the only queue that packets will experience in the switch. At each one of the possible 64 egress ports, an output queue is found, of 128 packet size, where packets are gathered before jumping to the output interface. As we are working on a virtual environment, after the egress queue, a final queue can also be found. This is the queue of the operating system virtual interface, which has been defined with a length of 1000. Although the important queue where scheduling is applied, and the one that we will be able to monitor and control is the one of the traffic manager, is extremely relevant to be aware of the two other queues that coexist in our experiment, to properly define their lengths and to understand the behaviors obtained.

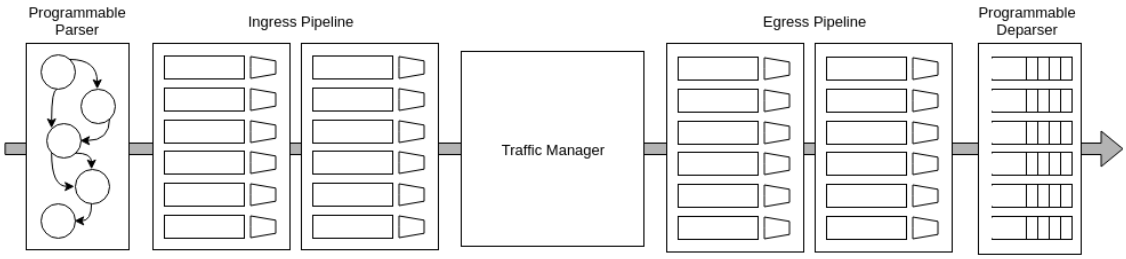


Figure 4.17: Protocol Independent Switch Architecture (PISA).

In Section 4.3 we mentioned the significance of being aware of the ingress pipeline bottleneck, to make sure that the congestion in our simulation scenarios were produced by the capacity, and therefore scheduling could take place in the traffic manager. Here, we not only insist on that, but we add the importance of making sure that queues in the egress and the output of the switch achieve blocking behavior, so that priority queues in the traffic manager can adapt their rates and priority scheduling can take place. Our experiments have shown that small queue sizes in the operating system queue, can make it execute drops instead of blocking, not letting traffic manager queues to adapt their rate, and most importantly, breaking the strict priority regime. If traffic manager queues can not perceive the bottleneck and adapt their transmissions accordingly, all the queues will forward their packets without limit, resulting in a high amount of packet drops in the output and a complete loss of PIFO performance. A great and easy way to verify the delay being experienced by packets according to the queue state, is to depart from the PIFO strict priority scenario, which we reviewed in Section 4.3.2. For a broader understanding, we monitored from the receiving end-host not only the queuing delay, but also the queues that each packet has encountered along the path, together with the rank it was assigned and the queue it was allocated.

Executing the experiment on the single-hop topology of Figure 4.1, by progressively transmitting 5.5Mbps UDP flows of increasing level of priority, we obtained the results of Figure 4.18. As can be seen at the beginning of the simulation, it requires certain time for the traffic manager queues to start building up and observe congestion, as the output and egress queues of 1000 and 128 packets respectively have to be filled before. After this first transient, congestion is perceived in the lower priority queue with the first flow transmission, and with packets encountering up to 64 other packets in queue. When second flow starts transmitting, the next priority queue will be congested as well, filling up to 128 packets and the same behavior will

repeat until completing the eight different queues. As can be also observed, although more and more packets occupy the levels of all strict priority queues, the highest priority flow will only perceive the packets of its own queue. Delay experienced by the packets transmitted, which will be the ones of the highest priority queue, will only see their transmissions affected by a maximum of 64 packets. Lower level queues will not have any effect on the delay perceived by the transmitting flow at each moment. Therefore, the maximum delay is on the order of 160ms, which results from the 64 packets encountered, each one of 1512Bytes given by the default *iperf* configuration, transmitted at 5Mbps rate. Note that due to the way queues have been designed, not blocking packets in an individual manner, instead of perceiving a constant value of 64 packets in queue, we experience continuous variations in the queue depth.

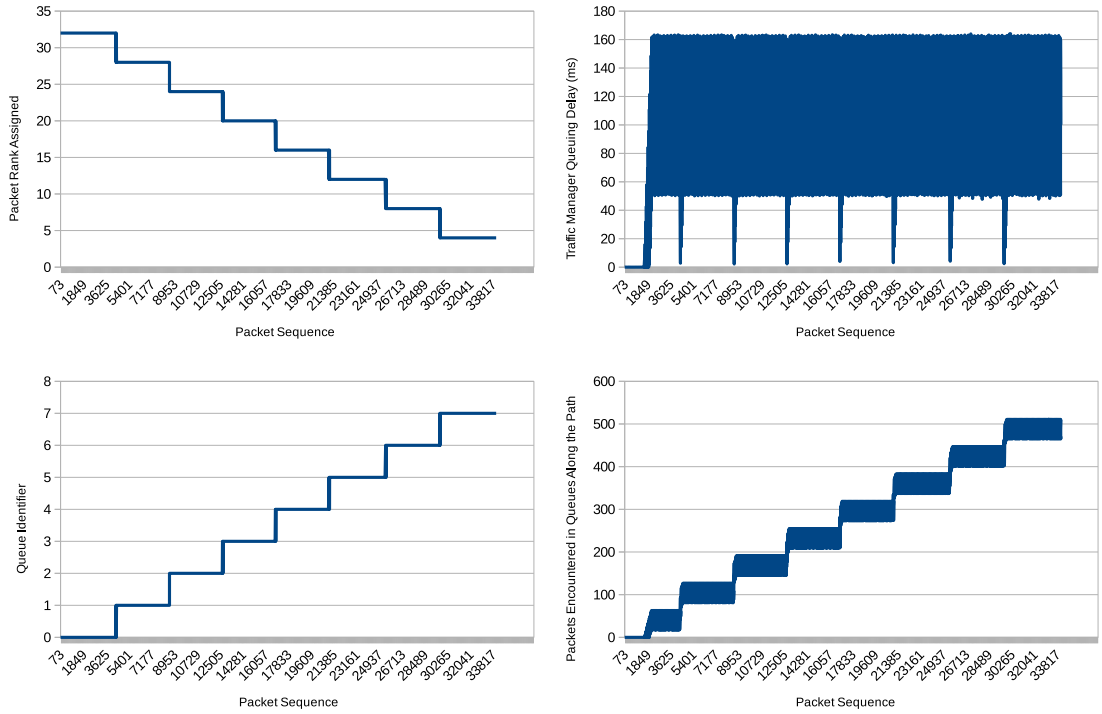


Figure 4.18: Queue evolution and delay under progressive higher priority flow generation.

4.4.1.2 Effects on a single path

Having analyzed the main components taking place in queuing delay, we can now dive into the effects of FIFO+ under a single flow transmission. Following the directives presented in the introduction to generate consecutive congestion in order to perceive FIFO+ effects, we will work under the scenario of Figure 4.19, but with link capacities of 5Mbps, 4.9Mbps, 4.8Mbps, 4.7Mbps and 4.6Mbps for (s1-s2),(s2-s3),(s3-s4),(s4-s5) and (s5-s6) respectively. A single flow will be transmitted from h1 to h6 experiencing the consecutively-decreasing bottleneck along the path. In reference to queue sizes, to compare FIFO and FIFO+ in a fair manner, we will allocate the same queue space for the two algorithms. Following the original work in [CSZ92], the queue size will be set to 120 packets for FIFO and the equivalent 8 queues of 15 packets for FIFO+. After transmitting five UDP flows from h1 to h6 of 1Mbps each, observed results are: an average delay of 815ms with a percentile 99.9 of 997ms for FIFO, and an average delay of 177ms with a 99.9 percentile of 465ms for FIFO+. In contrary to what we were expecting, not only the 99.9 percentile is decreased, but also the average delay. This is due to the fact

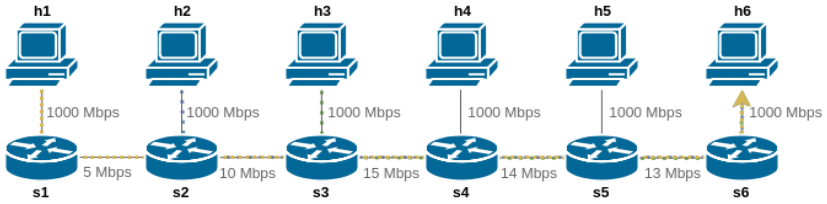


Figure 4.19: Multi-hop algorithms topology.

that our PIFO implementation divides the initial FIFO queue on 8 strict priority slices. When under UDP congestion FIFO+ is applied, only packets from the highest priority queue are being transmitted. Instead of the total queue number of 120 that packets experience in FIFO, in FIFO+ only the 8th part of those packets are affecting the final delay.

4.4.1.3 Effects in traffic from different paths

A more complex architecture using the two techniques to generate congestion, and observe the effects of FIFO+ when performing scheduling, is exactly depicted in Figure 4.19. On it, access links have been set to 1000Mbps, letting the connections among switches be the ones creating bottlenecks. Hosts 1, 2 and 3 will be responsible for generating traffic, congesting all links from s1 to s6. They will transmit following different types of distributions, all of them adding up a mean rate of 5Mbps. This way, traffic from h1 will first congest s1-s2 link. To it, traffic from h2 will be added in s2, to congest the link s2-s3. In s3, traffic from h3 will join, congesting s3-s4 as well, and finally, from s4 to s6, link capacities will be decreased progressively to make congestion take place until reaching the final destination. Concerning traffic distributions, three different simulations will be performed. On all of them, each host will be in charge of transmitting 5 flows. In the first simulation, flows will run constant UDP transmissions with packet departure rates of 125 packets per second and packet sizes of 1000Bytes. In the second simulation, each flow will still transmit packets of 1000Bytes, but this time under an exponentially distributed packet departure rate of average 125 packets per second. In the third simulation, a Pareto distribution will be used, of shape $\gamma = 2$ and scale $\delta = 4$, for a mean inter-departure time of 8ms, giving also the equivalent 125 packets per second average rate. [1]

The results obtained for each simulation are summarized in Table 4.2. In all cases, and in the same way that it was observed for the single-path experiment in Section 4.4.1.2, both the average and the tail delays are smaller in FIFO+ than in FIFO. Having already commented the queue slicing effect which causes this behavior, it is interesting to put the focus on how FIFO+ reacts when scheduling flows that have traveled different paths. As a quick remark, FIFO+ original idea was to just prioritize those packets that, within a given flow, had suffered higher waiting times in previous hops so that, at the end of the path, the tail delays of the flow could be in big part reduced, although increasing a little bit the average delay. However, what we encounter when applying FIFO+ on traffic flows from different paths is that, as all original deadlines are set with the same value, packets from flows having traveled longer distances will arrive to the colliding-switch with substantially smaller deadlines than closer flows. This will make FIFO+ not perform as it was originally expected, but just completely prioritizing packets from further sources over closer ones. This effect can be clearly observed in Figure 4.20. All packets, independently of whether they are transmitted from h1, h2 or h3 are initialized with

[1] Note that any Pareto probability distribution is characterized by two parameters: shape γ and scale δ, where the mean of the distribution is defined by: $\mu = \frac{\gamma \times \delta}{\gamma - 1}$.

FIFO+		
Constant	Avg. Delay	Percentile 99
Flows h1-h6	58	159
Flows h2-h6	55	158
Flows h3-h6	57	170
Overall	57	162
Pareto	Avg. Delay	Percentile 99
Flows h1-h6	49	166
Flows h2-h6	46	159
Flows h3-h6	52	171
Overall	49	165
Exponential	Avg. Delay	Percentile 99
Flows h1-h6	48	163
Flows h2-h6	46	154
Flows h3-h6	50	167
Overall	48	162

FIFO		
Constant	Avg. Delay	Percentile 99
Flows h1-h6	400	482
Flows h2-h6	245	297
Flows h3-h6	168	206
Overall	269	471
Pareto	Avg. Delay	Percentile 99
Flows h1-h6	310	456
Flows h2-h6	204	285
Flows h3-h6	152	203
Overall	221	439
Exponential	Avg. Delay	Percentile 99
Flows h1-h6	359	462
Flows h2-h6	229	287
Flows h3-h6	163	204
Overall	249	447

Table 4.2: Head tail-delay minimization performance for FIFO+ respect FIFO.

the same deadline values. However, when FIFO+ is applied in s2, packets from h1 have already crossed s1, and their deadlines have been updated, while packets from h2 still have their ranks at initialization values when performing the scheduling. H1-packets are therefore completely prioritized over packets coming from h2. The same happens in s3. Packets coming from h1 have already crossed s1 and s2, with the corresponding waiting times in each switch. Packets from h2 have crossed s2, and packets from h3 are still in initialization state. Therefore FIFO+ directly prioritizes h1 and h2 packets, with smaller deadlines, above packets coming from h3. By extending this behavior throughout the following switches, at the end of the path, all flows finish achieving equivalent delays.

While this behavior can be very beneficial for flows with packets having traveled longer distances, it can be seen as an unfair algorithm for flows crossing shorter paths. In order to achieve a fair FIFO+ implementation, deadlines in the end-hosts should be specified already taking into account which paths are aiming to be followed. For example, in our case, deadlines from h1 should be specified as greater than deadlines from h2. At the same time, deadlines from h2 should be configured bigger than the ones of h3, considering that packets coming from those sources will have to travel different distances (in terms of hops, and therefore waiting times in queues) before finding each other in the common path. This is the crucial reason, why scheduling algorithms in which deadlines have to be updated throughout the path, imply a high level of synchronization, sometimes very difficult to control.

4.4.2 Least Slack Time First: Optimizing deadlines from the end-host

LSTF has been demonstrated to be the most versatile algorithm, being able to closely replicate behaviors of diverse scheduling techniques, by just prioritizing packets based on slack deadlines specified from the end-hosts [MARS15]. One can easily observe that the "slack" definition in LSTF is equivalent to the "rank" definition in PIFO, or the "deadline" in EDF. Consequently,

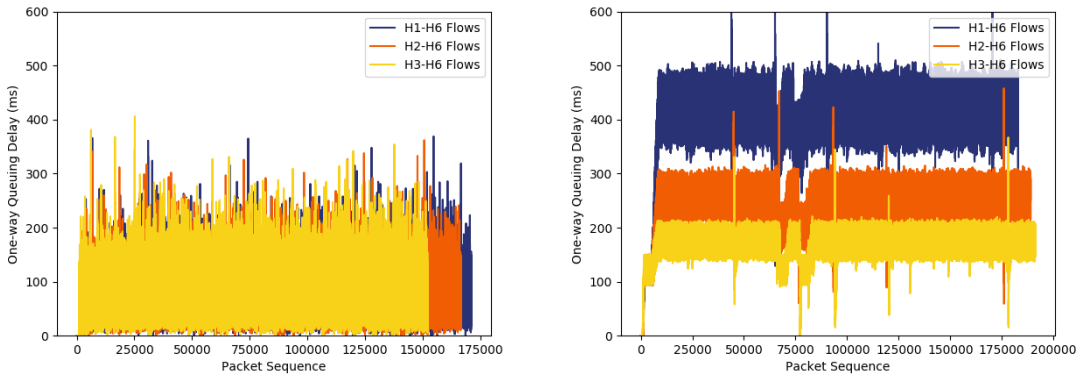


Figure 4.20: Queuing delay in FIFO+ and FIFO, experienced in traffic manager queues of consecutive switches along the path by constant 1Mbps transmissions UDP traffic generation.

LSTF can be understood as a slice of the PIFO abstraction considering only those algorithms that can be implemented by setting ranks from the end-host and updating them along the path. This excludes LSTF from being able to replicate algorithms like WFQ, which can not be described with a single end-host ranking definition. All multi-tenant scenarios, in which information from different entities is required for concerns like isolation or fairness, are not implementable on a system like LSTF. With PIFO, and therefore also with our solution, those limitations are vanished through rank definitions directly implemented from the switch, as has been already discussed in Section 4.3.4. The interesting aspect of LSTF, further away from its similarities with PIFO, is the flexibility that it provides at the time of configuring deadlines. As those ranks, or slacks, can be defined at a per-packet granularity, in some cases, better results than traditional fixed-rank algorithms can be achieved. For the sake of familiarity with the demonstration, we will consider as baseline the widely researched SFF, and we will see how LSTF is able to outperform it, providing smaller flow completion times under certain circumstances. LSTF can be easily understood in this case as a mix between SFF and FIFO+, and the obtained results completely reflect it.

As can be observed in Figure 4.21 and Table 4.3, LSTF implementation achieves in general very similar results to SFF. Following the same rationale than in FIFO+, we can see how the difference in deadlines of traffic having traversed different number of hops, impacts in a bad manner to flows from shorter paths. Those flows have to experience how all the packets from longer paths are prioritized over them, deriving in a worse performance than with the original algorithm. However, when focusing in traffic coming from h1, the action of updating deadlines with queuing times spent along the path, makes LSTF able to defeat SFF by minimizing even more the resulting average flow completion time.

Nonetheless, the positive results of LSTF, can not always be generalized. For illustrative purposes and under the idea that LSTF is optimal in minimizing the number of deadlines not fulfilled from the ones specified, Table 4.3 also shows the results when LSTF deadlines are defined following the SFF flow completion times from the previous experiment. As can be observed, satisfying a higher number of deadlines, does not necessarily mean that the global performance will be improved. If the deadlines left to be achieved are the ones with higher impact in the final metric of interest, LSTF will not be able to outperform the desired result. This exhibits the lesson again, that LSTF, as well as FIFO+, and in general all the algorithms which have been implemented on the PIFO abstraction, requires precise levels of detail when

SFF	
TCP Traffic	Avg. FCT
Flows h1-h6	46.38116
Flows h2-h6	**43.12075**
Flows h3-h6	**43.94172**
Overall	44.48121

LSTF (Flow Size)	
TCP Traffic	Avg. FCT
Flows h1-h6	**45.81735**
Flows h2-h6	45.59687
Flows h3-h6	45.45759
Overall	45.62394

LSTF (SFF Average Delay)	
TCP Traffic	Avg. FCT
Flows h1-h6	52.94233
Flows h2-h6	52.94233
Flows h3-h6	48.71241
Overall	52.43344

Table 4.3: Deadline definition strategies performance in Flow Completion Time (FCT).

setting up its configuration. A part of showing how LSTF can be used to improve some algorithms design, this example recalls on the idea that deadline management can sometimes demand high levels of accuracy and fine detailed knowledge of the topology, specially in cases where updates are performed along the path. Together with the fact that benefits observed when applying those changes are usually not that significant in the final performance, all these reasons make us believe that the complexity required for using this type of algorithms is not always worth the effort for most common scenarios.

4.5 Limitations and constraints

As it was stated in the original PIFO proposal, the new abstraction presented only allows the implementation of work conserving algorithms. This leaves out all those techniques which perform any type of rate modification or traffic shaping (non-work conserving) as well as hierarchical scheduling. The same exact restrictions apply to our strict priority approximation SP-PIFO. It is important to remark that the current version of P4 does not allow any non-work conserving policy, as the egress queues limitation can only be imposed by the output link bottleneck. While it is true that queues can have a rate limit specified, this limit can only be defined at compilation time, or through the control plane in software implementations. Individual queue rates can not be yet (according to the latest P4 version at the time of writing this report) dynamically modified by the P4 program directly. This lack of support for non-work-conserving algorithms derived from the P4 specification has been the one stopping us from further researching those techniques in which rate limiting and traffic shaping is performed from the switch. In case future enhanced versions of P4 might allow rate control for transmissions through P4 program definitions, to be executed at line-rate, a new range of schedulers including all type of token buckets would then be implementable.

Another interesting reflection is that PIFO is not a tool for preventing congestion by itself. SP-PIFO approximation is neither. With the proposed solution, congestion is not prevented and congestion is not eliminated, but congestion can be controlled. To eliminate congestion, strictly speaking, one would need to link PIFO to some mechanism capable of changing the paths that packets follow from source to destination. Although we would like to discuss this idea in future steps, and again, the higher the number of tools working together for a common purpose, the better the result that can be obtained, the presented version of SP-PIFO does not currently give answer to this type of scenarios. What SP-PIFO can do instead, is adapt packets priority in each switch, so that the negative effects of congestion can be minimized. By, for example, making sure that loss sensitive applications are protected in front of not delicate ones, the consequences of congestion can be controlled until the problem is dissolved. In this same example, if any rate-control was applied from the end hosts, users from applications with

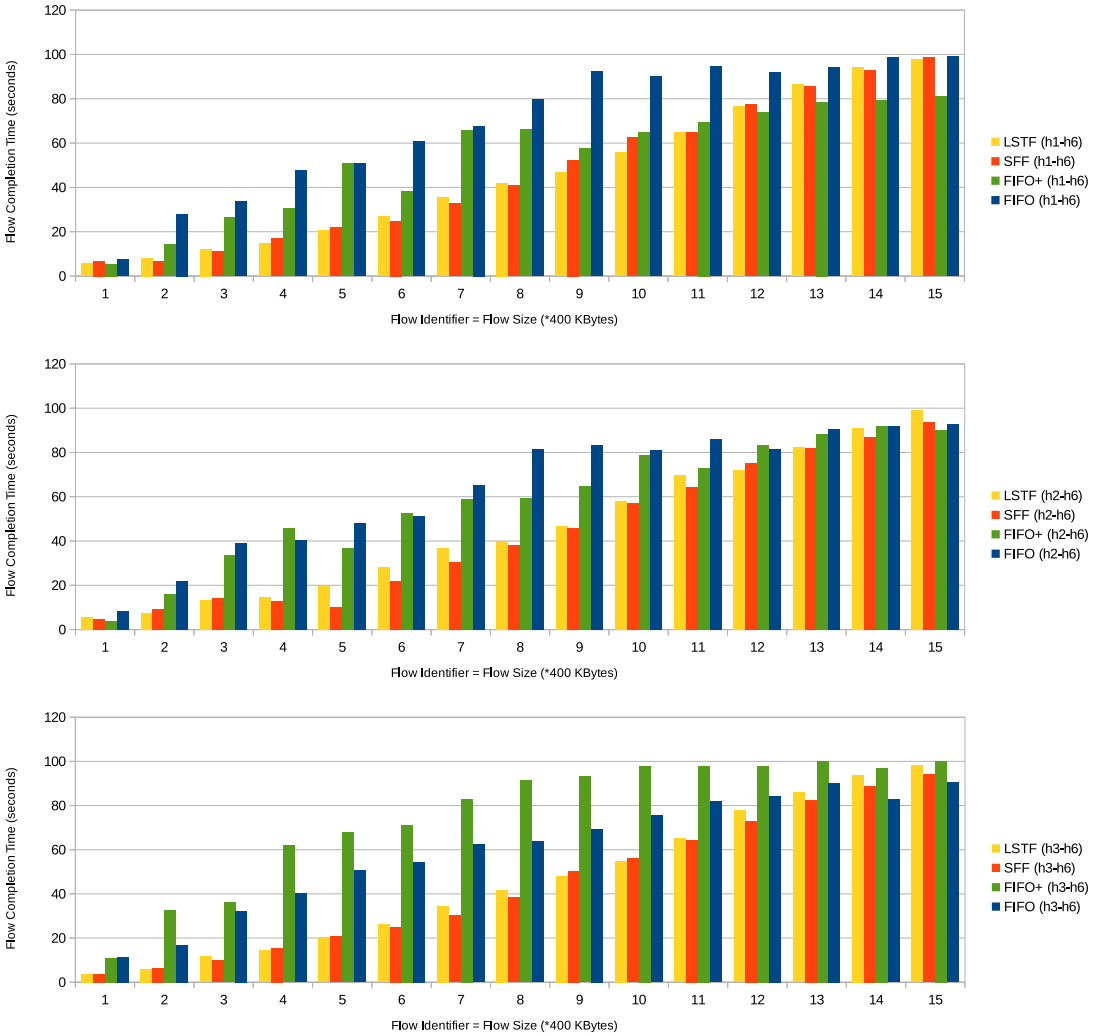


Figure 4.21: Flow Completion Time results for multi-hop scheduling techniques.

lower level of priority would stop receiving acknowledgements and would probably decrease their rate. As ranks can be dynamically updated, once congestion is solved, packet ranks can be restored to the default values and the initial scheduling behaviors can be easily recovered. It is important to remark that congestions are just events that occur in very particular scenarios, while the majority of time networks run under low utilization levels. Is just in those determinate moments, when quality of service management jumps in and plays such an important role. More advanced techniques aiming to fight congestion have been classified under the "active queue management" field umbrella. Most of them try to predict when congestion is about to happen and react by sending back notifications to the origin hosts, so that they can adapt their transmission rates. The others tend to directly control congestion from the switch by taking advantage of intrinsic traffic shapers or rate limiters at the input. Although, as mentioned, the second group of techniques are not possible due to the lack of traffic shapers in current programmable switches, traditional notification-based active queue management techniques like RED and ECN are perfectly implementable and completely compatible with our solution.

4.6 Implementation in real networks: Barefoot Tofino

One of the clearest requirements that we posed when starting the algorithm design, was making our solution implementable in real hardware. We discussed that most of proposals in nowadays literature relied on ns-3 software simulations or traffic control configurations, which, although providing great results, could be perceived as distant from what current infrastructure was actually able to realize. Following this idea, throughout our algorithm design, we have tried to keep the focus on what possibilities current network equipment was really offering. After having implemented and reviewed the performance of our scheme in software-simulations, we wanted to go a step further and demonstrate the feasibility of executing SP-PIFO in a real hardware switch. Conscious of the intrinsic difficulties that this requirement has supposed, in this section, we will summarize some of the challenges that we have encountered, together with the major insights that have helped us in defining a final accurate approach.

When P4 authors exposed their original idea, the first proposal was to make from it a target-independent high level programming language. One, that would easily represent new manners of processing packets within a common framework, and could then be directly deployed and executed in any P4-supporting switch. All this in a transparent manner, without having to know the intrinsic details of the underlying hardware, nor having to modify the program according to the specific target constraints. Each different switch would just have to define a specific compiler able to convert this P4 general description into the equivalent set of instructions that could finally be understood and executed by the particular selected chip. However, this initial (and quite idealistic) objective can be considered to be, still today, on its early stages. P4-supporting target manufacturers have not been able to provide compilers capable of directly translating P4-default programs into target-specific configurations. Instead, they provide alternative versions of P4 which already consider the most important hardware limitations, and are the ones that algorithm designers should use when aiming to translate their initial program descriptions to the final policies that the switch will be able to run.

In this thesis, we decided to work with Barefoot Tofino, a high-performance P4 switch that we are glad to have in our lab, and which has been proven to be the fastest one in the world [Bar16]. Although being fairly programmable, it is still some miles behind the state-of-the-art of P4 language evolution. In particular, Tofino is not compliant with the P4-16 latest specification (the one we have used throughout the thesis). It supports only P4-14 but, the latest Tofino-P4 compiler is still not able to directly convert original P4-14 programs into the equivalent set of instructions for the definitive pipeline configuration. Instead, same as happens with the rest of P4-compliant targets, a restricted and modified version of P4-14 is provided, which considers all the target physical restrictions and constraints that appear when working in real silicon implementations. This specific version of Tofino-P4, has shown us that even working with the behavioral model can sometimes be too optimistic in defining what will actually be implementable later on in hardware. To a certain extent, a common mistake when considering this new wave of programmable networks (and so it was our), is to think of programmable switches as forwarding CPUs, with great saving state capabilities and extended computation facilities. Reality is, however, that programmable switches are just forwarding instances which allow a narrow degree of flexibility. Even our program which, at first, seemed to be reasonably simple to implement, had to be re-designed in order to accommodate the final hardware limitations.

The following content has been hidden for confidentiality purposes. Contact the author for further information in case of having explicit written permission by Barefoot Networks Inc.

4.7 SP-PIFO for predictable networks. Next steps and open challenges

Having analyzed how effectively SP-PIFO can be used to make the traffic manager programmable and adapt to the different performance objectives, the consequent question arises: Now, what? Now that we have made the traffic manager programmable, with SP-PIFO providing flexibility, how can all this knowledge be applied? How should SP-PIFO be used? It is clear, that making the last remaining pipeline of the PISA architecture also programmable, opens a ton of new possibilities. The starting motivation of this thesis was moving towards predictable networks. Making scheduling programmable was a fundamental milestone to overcome. After having finished this thesis, with the objective of making scheduling programmable achieved, it is interesting to take some time and understand how our proposed solution can be used from now on. The next step towards network predictability is, without any doubt, bringing our SP-PIFO solution to the network perspective. Throughout this thesis we have been working under the assumption of a simple topology, aiming to achieve specific requirements. From now on, this focus should change. Instead, a whole network should be considered, in which the equipment is shared by a given workload where different performance objectives coexist. By itself, working on a global network instead than on a switch level, generates new questions to be discussed. If we add the fact that traffic flows sharing these resources do not aim to achieve the same type of requirements, that opens a wide range of challenges needed to be solved. In the new scenario, where the whole network is covered with SP-PIFO capable switches, the following 5 principal aspects need to be clearly considered.

- **How to correctly set the ranks:** We have seen some examples for a single switch and a single objective. But in a network where not only multiple switches are considered but also a diverse group of objectives, the challenge of how to combine those immediately arises. Can we draw multiple paths? Are the requirements mutually exclusive? In some occasions it may be possible to partition the network following requested demands. But most probably, in the vast majority of cases this isolation of requirements is not feasible in practice. We will then need to adapt the SP-PIFO policies for the global best of the network. Those policies, will not necessarily have to be constant throughout the path. One can clearly imagine that delay-sensitive flows will probably have to cross switches giving them maximum priority but also other hops potentially implementing fairness or other types of algorithms. Which are the optimal SP-PIFO configurations to use across the network, to serve a given workload at each moment? This is the first question that the final system will definitely need to answer.

- **How to define and evaluate the application requirements:** In order to know which are the best policies for a given workload distribution, is important to bring back the problem of how application requirements are specified and where do they come from. It is interesting to make here the reflection of probably the main reason why quality of service has not been successful up to now, which is the hazardous configuration of all the parameters that end-users were forced to manipulate in order to implement it. Tweaking all those parameters supposed a too high complexity, which stopped the big majority of users from using these techniques. In our view, the network should be the one in charge of adapting the services by itself, without having to depend on end-users responsibility to properly define the final strategies to follow. It is a task of the network to automatically configure the best polices, without need for end-users to even know

what is going on at each moment. No need to configure parameters, no need to define requirements. The network should be able to determine what is good or bad for the application. For instance, a promising idea would be to perform load balancing of traffic flows through different paths, measuring the success of each option at the end and using this learning to optimize the best arrangement for future allocations. And this uncovers the next challenge to consider, which is how to correctly define application requirements and evaluate their fulfillment. Since the beginning of the thesis, it was made clear that the network should face the dynamically changing application flow requirements. It was seen that these requirements could be decoupled in jitter, delay and throughput and that they were attached to a desired certain level of quality experience. Quality of experience (QoE) is not an objective metric that can be quantified by the network. Is, instead, a perception by the end-user about how good or bad the received service was. Making the network aware of the final quality of experience received by the user is a key challenge that needs to be solved. And probably the most important one, as it is the basis for all the predictable networks paradigm. How can the user perception of a service be linked to a determined rank distribution? We require means for the network to have access to this information in a transparent way from the user perspective, to evaluate service performances and requirements for each type of connection. Some services like Skype calls, already provide mechanisms for the user to rate the quality of service experienced after this having taken place. It would be reasonable for us, to think about each service having an internal API, giving feedback to the network about the satisfaction of the user from the service. This API should transform users rating at high-level to low-level evaluation metrics that could be easily understood by the network to determine if the allocated policies were the correct, to train the learning system, and adapt future allocations with the knowledge and experience obtained. One possibility to have in mind, should be to think about an available API for each type of application, that sends feedback to the network according to the perception that the user had from the received service under the given conditions. This feedback could be used to adapt the ranking policies accordingly, while storing all this information in the system for performing more intelligent decisions in the future and achieving better results.

- **How to properly design a rank-setting architecture:** Significant emphasis has to be put on describing a clear rank-setting architecture, defining both the appropriate entity (or entities) in charge of developing SP-PIFO rank policies, and the one (or ones) in charge of implementing those policies. Two main options are clearly presented: taking advantage of the centralized vision of a controller in an SDN-like philosophy, or applying a distributed paradigm. Both type of architectures have pros and cons. While a centralized architecture reaches a higher level of optimization thanks to the complete view of the network, it suffers from the single point of failure problem. At the same time, distributed solutions, less optimal but more robust to failures, could also suppose a big amount of overhead. Concerning where to execute those policies we have two main options. The first one is following what we have been doing in the thesis and letting end-hosts select the desired rank distribution. The other alternative, if a higher level of control is desired, would be to require access-switches to define those ranks. In this same group of challenges, another aspect to consider is how the policies can be mixed to achieve different objectives. Are those objectives mutually exclusive? As now a whole network is considered, not a single switch, we can think of paths where different rank definitions are used at each path. A certain flow may cross some switches implementing Weighted Fair Queuing, others Shortest Flow First and some others with just FIFO behav-

ior. Performance objectives have to be defined end-to-end and not at a per-hop level anymore. As a starting point, it is always easier to think about the centralized solution, with the controller optimizing the resources and spreading the policies through APIs with the style of OpenFlow. In the distributed case, instead, data plane packets should be the ones allowing switches to communicate with each other to optimize the SP-PIFO policies throughout the network. One could think of some type of feedback, used to communicate the final results of a specific rank arrangement, to be transmitted from the destination back to the origin, so that switches along the path could use the result information to update the polices accordingly. A simple example for the sake of understanding would be the one for delay. Assume that when a packet reached the destination, a notification could be generated if the achieved delay exceeded the expectations. With this notification, switches could set higher priority ranks for the same type of packets in future communications to increase the chances of the requirements being fulfilled. In this distributed version, two options could also be distinguished. The first would be letting the end-hosts specify the ranks for their connections. The other would be controlling the ranking definition from access-switches on the network. Depending on the desired level of control or independence of the end hosts, one or the other option could be applied.

- **How to synchronize rank-definitions along the path:** As was already envisioned in subsection 4.4.1.3, for algorithms like FIFO+ in which rank definitions are dependent of the paths followed by each packet, when flows having traversed multiple paths encounter, their rank definitions can collide. A mechanism considering this type of problems should also be clearly considered.

- **How to accommodate traffic dynamism:** While all the previous challenges can be studied with a static workload in the network, to bring the problem a bit forward, one should consider including the dimension of time. Assuming that the network does not have a fixed workload to serve, but needs to accommodate the dynamically changing traffic flows with varying requirements, for any types of architectures defined, the proper predictable network design should include means to update and adapt to the new demands that traffic could generate. This supposes not only ways to update the ranking policies along the switches, as the requirements distribution change over time, but also the need of defining admission policies to control whether new demands should be preempted or not, respect existing traffic flows on the network, for the global and fair benefit of the final users.

All those questions - and more - will have to be answered in detail when trying to develop accurate and sophisticate solutions based on the new predictable networks paradigm. What it is clear, however, is that tools like SP-PIFO, after having proved the great versatility that it can adopt, and the wide range of results that it can achieve, will be key in enabling new paths and enforcing research to keep progressing towards that direction. The flexibility with which the proposed solution has been designed, together with the diverse implementation issues deeply discussed throughput this thesis, will make the transition to the forthcoming next generation predictable networks much more straightforward to realize.

5 Conclusions and Future Work

In this Master thesis we have focused on improving current networks quality of service, by taking advantage of the latest advances in programmable data planes. Having defined and modeled our final objective with the idea of predictable networks, defined as those systems able to anticipate, plan, adapt, update and react to the dynamically changing traffic demands of application flows, we have realized that the principal steak-holder sustaining this paradigm is packet scheduling. Despite the great evolution that scheduling has experienced in recent years, and the amount of efforts it has gathered from the research community, not a single algorithm has been shown to outperform the others in all types of scenarios and use-cases. At the same time, promising abstractions have lately been proposed to make traffic managers more programmable. Special attention has been received by PIFO, which realizes that every scheduling algorithm can be defined in two main parts: a ranking definition which describes the position that each packet should occupy in queue, and a push-in first-out structure, that executes the defined policies by enqueuing packets according to the ranks. However, the amount of time required to implement those architectures on hardware, has made it not feasible up to today, to enjoy a final programmable scheduling solution.

Motivated by the urgent need of those programmable structures, in this thesis we have proposed an alternative novel approach. Based on using traditional strict priority schemes, available in currently deployed hardware equipment, the proposed SP-PIFO selects the best position to place each packet in an adaptive and dynamic manner, being able to give response to a broad range of rank configurations without previous traffic knowledge required in advance. SP-PIFO has been shown to correctly approximate a wide variety of performance objectives, from minimizing flow completion times, to reducing tail packet delays or distributing bandwidth shares following max-min fairness. By using strict priority queues as a baseline, SP-PIFO is not only implementable for the new wave of programmable switches but also on already deployed general chips on our networks. A side of the simulations performed on virtual environments for testing purposes, the proposed algorithm has also been implemented and tested in the Barefoot Tofino hardware target, able to execute it at line-rate achieving output speeds up to 6.5Tbps.

Although SP-PIFO has been conceived as just an approximation of the idealistic hardware solution, we strongly believe that its main ideas can not only play an important role in the transient time until a built-in design is proposed, but it will help, as it has done for us throughout this thesis, to outline and investigate the main limitations and restrictions that this kind of queue structures will intrinsically have. Being a key tool for enforcing research on programmable scheduling in the following years, it will be our perfect partner to keep moving on our search towards the next generation of predictable networks.

Bibliography

[AGM+10] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan, *Data center tcp (dctcp)*, ACM SIGCOMM computer communication review, vol. 40, ACM, 2010, pp. 63–74. 12

[AKE+12] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda, *Less is more: trading a little bandwidth for ultra-low latency in the data center*, Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2012, pp. 19–19. 12

[AYS+13] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker, *pfabric: Minimal near-optimal datacenter transport*, SIGCOMM Comput. Commun. Rev. **43** (2013), no. 4, 435–446. 12, 39

[Bal16] Hari Balakrishnan, *Scheduling for fairness: Fair queueing and csfq*, MIT, Computer Networks, Scheduling, Isolation, and Fairness Lecture, Fall 2016 (6.829) (2016). 43

[Bar16] Barefoot, *The world's fastest and most programmable networks*, White Paper (2016). 60

[BDG+14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker, *P4: Programming protocol-independent packet processors*, SIGCOMM Comput. Commun. Rev. **44** (2014), no. 3, 87–95. 3

[BDP12] Alessio Botta, Alberto Dainotti, and Antonio Pescapè, *A tool for the generation of realistic network workload for emerging networking scenarios*, Computer Networks **56** (2012), no. 15, 3531–3547. 40

[CSZ92] David D. Clark, Scott Shenker, and Lixia Zhang, *Supporting real-time applications in an integrated services packet network: Architecture and mechanism*, SIGCOMM Comput. Commun. Rev. **22** (1992), no. 4, 14–26. 51, 54

[DKS89] A. Demers, S. Keshav, and S. Shenker, *Analysis and simulation of a fair queueing algorithm*, Symposium Proceedings on Communications Architectures &Amp; Protocols (New York, NY, USA), SIGCOMM '89, ACM, 1989, pp. 1–12. 9, 45

[Gol94] S. J. Golestani, *A self-clocked fair queueing scheme for broadband applications*, INFOCOM '94. Networking for Global Communications., 13th Proceedings IEEE, Jun 1994, pp. 636–646 vol.2. 10, 45

[HCG12] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey, *Finishing flows quickly with preemptive scheduling*, Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (New York, NY, USA), SIGCOMM '12, ACM, 2012, pp. 127–138. 11

[HG86] E. L. Hahne and R. G. Gallager, *Round robin scheduling for fair flow control in data communication networks*, NASA STI/Recon Technical Report N **86** (1986). 9, 44

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown, *A network in a laptop: rapid prototyping for software-defined networks*, Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, ACM, 2010, p. 19. 31

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner, *Openflow: enabling innovation in campus networks*, ACM SIGCOMM Computer Communication Review **38** (2008), no. 2, 69–74. 3

[MARS15] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker, *Universal packet scheduling*, Proceedings of the 14th ACM workshop on hot topics in networks, ACM, 2015, p. 24. 13, 56

[McK90] P. E. McKenney, *Stochastic fairness queueing*, INFOCOM '90, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. The Multiple Facets of Integration. Proceedings, IEEE, Jun 1990, pp. 733–740 vol.2. 10

[McK17] Nick McKeown, *Why does the internet need a programmable forwarding plane*, Networking Field Day 14, Silicon Valley, USA (2017). 3

[Nag87] J. Nagle, *On packet switches with infinite storage*, IEEE Transactions on Communications **35** (1987), no. 4, 435–438. 8

[NJ12] Kathleen Nichols and Van Jacobson, *Controlling queue delay*, Communications of the ACM **55** (2012), no. 7, 42–50. 27

[OS15] Kay Ousterhaut and Ion Stoica, *Slides on pfabric: Minimal near-optimal datacenter transport*, UC Berkeley, Big Data System Research: Trends and Challenges Class (CS294) (2015). 43

[PG93] Abhay K. Parekh and Robert G. Gallager, *A generalized processor sharing approach to flow control in integrated services networks: The single-node case*, IEEE/ACM Trans. Netw. **1** (1993), no. 3, 344–357. 9, 44

[POB+15] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal, *Fastpass: A centralized zero-queue datacenter network*, ACM SIGCOMM Computer Communication Review **44** (2015), no. 4, 307–318. 4

[Sah08] Anirudha Sahoo, *Packet scheduling slides (it610), quality of service in networks (cs 680)*, Kanwal Rekhi School of Information Technology (KReSIT), Indian Institute of Technology, Bombay (2008). 10, 45

[Sem01] Chuck Semeria, *Supporting differentiated service classes: queue scheduling disciplines*. 9, 44

[SSA+16] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown, *Programmable packet scheduling at line rate*, Proceedings of the 2016 ACM SIGCOMM Conference (New York, NY, USA), SIGCOMM '16, ACM, 2016, pp. 44–57. 15, 16

[SV95] M. Shreedhar and George Varghese, *Efficient fair queueing using deficit round robin*, SIGCOMM Comput. Commun. Rev. **25** (1995), no. 4, 231–242. 9

[SWSB13] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan, *No silver bullet: Extending sdn to the data plane*, Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks (New York, NY, USA), HotNets-XII, ACM, 2013, pp. 19:1–19:7. 13

[VHV12] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar, *Deadline-aware datacenter tcp (d2tcp)*, ACM SIGCOMM Computer Communication Review **42** (2012), no. 4, 115–126. 12

[WBKR11] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron, *Better never than late: Meeting deadlines in datacenter networks*, ACM SIGCOMM Computer Communication Review **41** (2011), no. 4, 50–61. 11

[Zho18] Xiaobo Zhou, *Packet switching networks and algorithms*, Coursera, Computer Communications Specialization, University of Colorado (2018). 8